ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизации обработки информации (АОИ)

Ю.Б. Гриценко

СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Учебное пособие

2017

Гриценко Ю.Б.

Системы реального времени: Учебное пособие. — Томск: Томский университет систем управления и радиоэлектроники, 2017. — 253 с.

Рассмотрены вопросы организации и построения систем реального времени. Сделан обзор современных автоматизированных информационных систем управления технологическими процессами и операционных систем реального времени используемых в них. Основное внимание уделено вопросу построения операционной системы реального времени QNX (QNX Software Systems Limited).

Предназначено для студентов направлений 09.03.04, Программная инженерия, 09.04.04, Программная инженерия, Методы и технологии индустриального проектирования программного обеспечения

Содержание

Предисловие

В учебном пособии рассказывается о системах реального времени. Приводится определение систем реального времени и их классификация, рассматриваются основные параметры и механизмы, обеспечивающие требования реального времени. Сделан упор на обзор операционных систем реального времени,

Базовыми категориями в освоении данного курса являются понятия и концепции: построения систем реального времени, диспетчеризация потоков, установка уровней приоритетов, механизмы синхронизации и защиты от инверсии приоритетов.

Для изучения дисциплины «Системы реального времени» необходимо иметь навыки программирования на языке высокого уровня Си, освоить курс «Операционные системы».

Учебное пособие по курсу «Системы реального времени» представлено в семи разделах.

В первом разделе вводятся определения систем реального времени, области применения, и вычислительные платформы систем реального времени. Также в разделе уделено внимание обзору архитектур операционных систем реального времени и рассмотрено типичное строение системы реального времени.

Второй раздел содержит описание развития автоматизированных систем управления технологическими процессами, назначения их отдельных компонент, а также представлены функциональные возможности SCADA-систем, контроллеров и технологических языков программирования.

Третий раздел включает вопросы организации операционных систем реального времени их функциональные требования и архитектуры построения.

Стандарты на создание операционных систем реального времени рассмотрены в четвертом разделе. Раздел содержит описание стандартов SCEPTRE, POSIX, DO-178B, ARINC-653, OSEK и стандарты безопасности.

Пятый раздел содержит классификацию операционных систем реального времени в зависимости от происхождения и обзор основных функциональных возможностей конкретных операционных сисетм реального времени

В шестом разделе приведено описание общего представление об ОС QNX, основные характеристики микроядра ОС QNX, описание связей между процессами, вопросы сетевого взаимодействия, механизм планирования процессов и первичной обработки прерываний.

В седьмом разделе рассматривается работа администратора процессов и управление ресурсами ОС QNX. Вводится понятие администраторов ресурсов, описываются следующие ресурсы — файловые системы, инсталляционные пакеты, символьные устройства ввода/вывода, сетевая подсистема QNX, графический интерфейс пользователя, печать.

1. Введение в системы реального времени

1.1. Определения систем реального времени

Понятия «реальное время», «операционные системы реального времени» известны всем, но толкуются они часто поразному и спектр этих толкований очень широк. Количество иллюзий и мифов в мире реального времени велико. Прежде чем перейти к их рассмотрению необходимо дать понятие определению параллельные системы.

Программные системы, которые по своему назначению должны обрабатывать одновременные события или управлять одновременно выполняемыми операциями, инициируемыми внешними по отношению к ним программами или пользователями, являются параллельными по своей природе [1].

К **параллельным системам** относятся:

– ***Системы реального времени*** (СРВ) и встроенные системы (специального назначения).

– Обычные и распределенные операционные системы (их компоненты распределены по нескольким компьютерам).

– Системы управления базами данных и системы обработки транзакций.

– Распределенные сервисы прикладного уровня.

Рассмотрим первый класс параллельных систем — системы реального времени.

Английский термин «real-time» и соответствующее ему в русском языке понятие «реальное время» является наиболее спорным и сложным термином. Данное понятие применяется в различных научно-технических областях и подразумевает некие действия, продолжительность которых определяется внешними процессами.

Специфическая особенность **систем реального времени заключается в том, что к ним предъявляются строгие временные требования, диктуемые окружением или определяемые ее назначением** [1].

Вышеприведенное определение не является единственным, для понимания смысла данного понятия, приведем еще несколько определений:

– *Система называется системой реального времени, если правильность ее функционирования зависит не только от логической корректности вычислений, но и от времени, за которое эти вычисления производятся. То есть для событий, происходящих в такой системе, то, КОГДА эти события происходят, так же важно, как логическая корректность самих событий* [2].

– *Реальное время (программное обеспечение): Относится к системе или режиму работы, в котором вычисления проводятся в течение времени, определяемого внешним процессом, с целью управления или мониторинга внешнего процесса по результатам этих вычислений* (IEEE 610.12 — 1990).

– *Системы реального времени — это системы, которые предсказуемо (в смысле времени реакции) реагируют на непредсказуемые (по времени появления) внешние события* [3].

Одной из функций таких систем может быть выполнение определенных действий в ответ на сигналы тревоги, и очень важно, чтобы они отвечали на них с определенной скоростью. В связи с этим существует разделение систем реального времени на два типа:

1. Системы с жесткими временными характеристиками — системы ***жесткого реального времени***.
2. Системы с нежесткими временными характеристиками — системы ***мягкого реального времени***.

Системой ***жесткого*** реального времени называется система, где неспособность обеспечить реакцию на какие-либо события в заданное время является отказом и ведет к невозможности решения поставленной задачи. Многие теоретики ставят здесь точку, из чего следует, что время реакции в жестких системах может составлять и секунды, и часы, и недели. Однако большинство практиков считают, что время реакции в системах жесткого реального времени должно быть все-таки минимальным. Большинство систем жесткого реального времени являются системами контроля и управления. Такие СРВ сложны в реа-

лизации, так как для них предъявляются особые требования в вопросах безопасности.

Точного определения для ***мягкого*** реального времени не существует, поэтому отнесем сюда все СРВ, не попадающие в категорию жестких. Так как система мягкого реального времени может не успевать ВСЕ делать ВСЕГДА в заданное время, возникает проблема определения критериев успешности (нормальности) ее функционирования. Вопрос этот совсем не простой, так как в зависимости от функций системы это может быть максимальная задержка в выполнении каких-либо операций, средняя своевременность обработки событий и т.п. Более того, эти критерии влияют на то, какой алгоритм планирования задач является оптимальным.

Еще одной важной характеристикой системы реального времени является ее природа — ***статическая*** или ***динамическая***[1]. В статической системе функционирование прогнозируемо и может быть определенно на этапе проектирования. В динамической системе запросы поступают нерегулярно и непредсказуемо, но система должна динамически отвечать на них с гарантированной скоростью.

Следует отметить, что понятие ***функционировать в реальном времени*** отнюдь не означает ***очень быстро*** — его суть заключается в том, что к системе предъявляются определенные временные требования, и они должны соблюдаться [1].

Приведем требования, сформулированные Д. Бэконом и Т. Харрисом [1], к системе реального времени:

– Необходима поддержка выполнения отдельных задач. Одни из них, такие как сбор данных, могут быть периодическими, другие, в том числе реакция на сигналы тревоги, — непредсказуемыми.

– Для каждой задачи могут существовать специфические требования, в частности точно определяющие время ее выполнения.

– Отдельные выполняемые системой задачи могут быть частью одной общей задачи — в таком случае производимые в их рамках действия должны быть четко согласованными.

По типу применения системы реального времени можно разделить на ***специализированные*** и ***универсальные.***

Специализированной СРВ называется система, где конкретные временные требования определены. Такая система должна быть специально спроектирована для удовлетворения этих требований. Обычно такие системы применяются, где есть риск человеческого фактора.

Универсальная СРВ должна уметь выполнять произвольные (заранее не определенные) временные задачи без применения специальной техники. Разработка таких систем, безусловно, является самой сложной задачей, хотя обычно, требования, предъявляемые к таким системам, мягче, чем требования для специализированных систем.

1.2. Области применения и вычислительные платформы СРВ

В течение длительно времени основными потребителями СРВ были военная и космическая области. Сейчас ситуация изменилась, и СРВ можно встретить даже в товарах народного потребления.

Основные области применения СРВ:

- ***Военная и космическая области***:
  - бортовое и встраиваемое оборудование;
  - радары, системы измерения и управления;
  - цифровые видеосистемы, симуляторы;
  - ракеты, системы определения местоположения и привязки к местности.
- ***Промышленность***:
  - автоматические системы управления производством; автоматические системы управления технологическими процессами;
  - автомобилестроение: симуляторы, системы управления мотором, автоматическое сцепление …
  - энергетика: сбор информации, управление данными и оборудованием …
  - телекоммуникации: коммуникационное оборудование, сетевые коммутаторы, телефонные станции …
  - банковское оборудование: банкоматы …
- ***Товары широкого применения***:

- мобильные телефоны;
- цифровое телевидение: мультимедиа, видеосервисы, цифровые телевизионные декодеры …
- компьютерное и офисное оборудование.

Рассмотрим более подробно применение систем реального времени в наиболее интересных областях, которые приведены в работе Д. Бэкона и Т. Харриса [1]:

Управление технологическим процессом. В компьютерных системах управление технологическими процессами осуществляется путем сбора и анализа данных, получаемых с помощью специального контрольного оборудования (рис. 1.1).

Речь может идти как о простых действиях, таких как, об измерении температуры и давления через заданные промежутки времени и сравнении полученных показателей с предельными значениями, так и о более сложных, например о сборе большого количества разнообразных данных, их математическом анализе и выдаче команд управления технологическим процессом на исполнительные механизмы. Точность, с которой работает подобная система, зависит от объема собираемых данных и времени, затрачиваемого на их анализ. Сбор и анализ информации производится с известной периодичностью, зависящей от нужд контролируемого или управляемого процесса. Так определение температуры в доменной печи, требует выполнения замеров каждые несколько секунд, а уровень воды в водохранилище достаточно проверять один раз в час.

Системы должны не только периодически выполнять определенные действия, но и реагировать на события, возникающие в непредсказуемые моменты времени, например, повышение температуры ядерного реактора или давления газа в угольной шахте.

Также может быть еще один вид непериодических действий системы — высокоуровневое регулирование параметров, осуществляемое по запросу управляющего персонала на основе общей картины процесса, например изменение количества выпускаемой продукции. Подобного рода действия не столь срочные, как, реакция на сигнал тревоги, но являются неотъемлемой частью общего процесса функционирования системы.

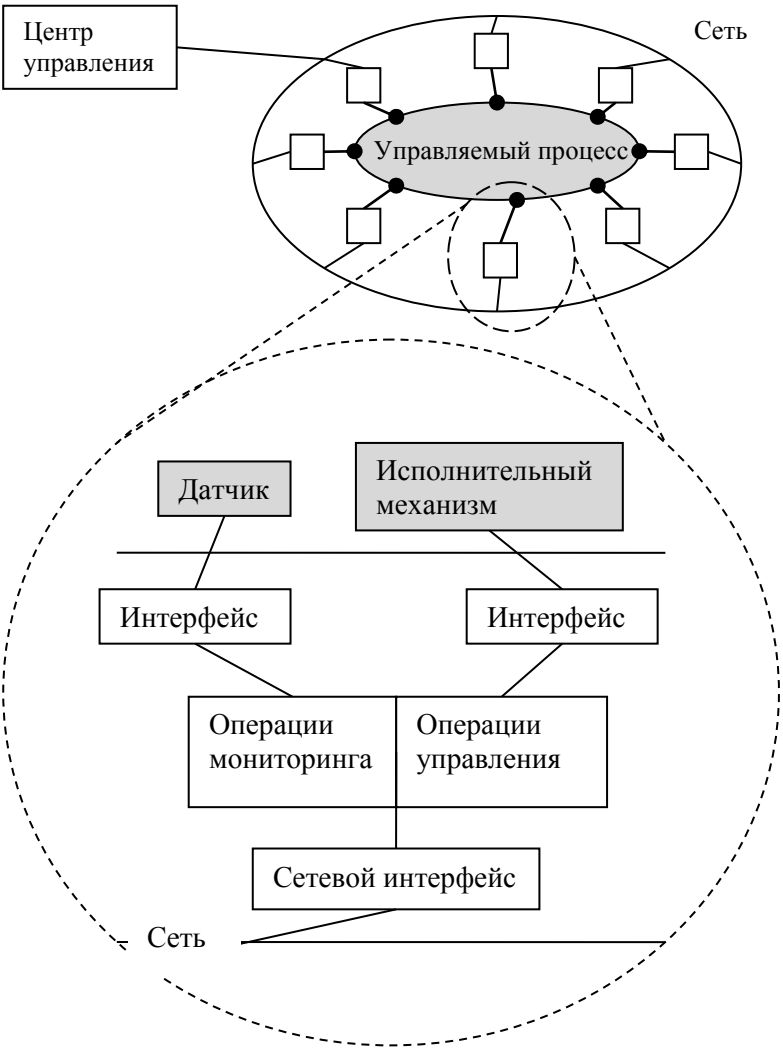

Рис. 1.1. Пример распределенной системы управления технологическим процессом

Описанные системы реального времени относятся к категории статических и имеют жесткие временные характеристики. Обычно для них разрабатывают периодическую схему, предусматривающую возможность поступления сигнала тревоги один за указанный период.

Поддержка мультимедиа. Мультимедийные приложения используются в самых разнообразных областях человеческой деятельности. Особенностью мультимедийных приложений является общее требование: два потока данных (звук и видео) должны воспроизводится синхронно и с необходимой скоростью.

Характерно, что запросы на воспроизведение данных мультимедиа поступают от пользователя в непредсказуемые моменты времени, возможно, параллельно с работой другого программного обеспечения. Нужно заметить, что видеоданные имеют очень большой объем, а следовательно, для их доставки необходима высокая пропускная способность соединений. Так, минутный видеоклип занимает 12 Мбайт памяти и должен передаваться со скоростью 200 Кбит/с. Доставка должна производиться равномерно — лишь при этом условии движение на экране будет выглядеть плавным и естественным, без рывков и резкой смены кадров. Таким образом, независимо от местоположения источника данных и загруженности системы к последней предъявляются требования гарантированного качества обслуживания. Иными словами, она должна быть способна воспроизводить мультимедийные данные с нужной скоростью, синхронно и без существенных потерь.

Мультимедийные рабочие станции можно определить как системы реального времени с нежесткими временными характеристиками: они должны работать в реальном времени, но для них допустимы некоторые задержки и снижение качества.

От обычных систем с разделением времени мультимедийные системы отличаются более высокими показателями вычислительной мощи, ширины полосы пропускания сетевых соединений, объема постоянной и основной памяти, а также программным обеспечением, способным эффективно использовать все эти технические возможности.

Согласно определению, СРВ должна обеспечить требуемый уровень сервиса в заданный промежуток времени. Этот промежуток времени задается обычно периодичностью и скоростью процессов, которым управляет система. Приблизительное время реакции в зависимости от области применения СРВ показано в таблице 1.1.

Таблица 1.1. Приблизительное время реакции в зависимости от области применения СРВ

Область применения СРВ	**Время реакции СРВ**
Математическое моделирование	несколько микросекунд
Радиолокация	несколько миллисекунд
Складской учет	несколько секунд
Управление производством	несколько минут

Видно, что времена очень разнятся и накладывают различные требования на вычислительную установку, на которой работает СРВ.

Часто СРВ существуют в нескольких вариантах, например, в полном и сокращенном, когда объем системы составляет несколько килобайтов.

Вычислительные установки, на которых используются СРВ, можно разделить на следующие платформы:

«Обычные» компьютеры. По логическому устройству совпадают с настольными компьютерами. Аппаратное устройство несколько отличается. Для обеспечения минимального времени простоя в случае технической неполадки процессор, память и другие элементы размещаются на съемной плате, вставляемой в специальный разъем так называемой «пассивной» платы. В другие разъемы этой платы вставляются платы периферийных контролеров и другое оборудование. Сам компьютер помещается в специальный корпус, обеспечивающий защиту от пыли и механических повреждений. В качестве мониторов используются жидкокристаллические дисплеи иногда с сенсорочувствительным покрытием.

Основное доминирующее положение на этих компьютерах занимают процессоры Intel 80x86.

Подобные вычислительные системы обычно не используются для непосредственного управления промышленным или иным оборудованием. Они служат как терминалы для взаимодействия с промышленными компьютерами и встроенными контролерами, для визуализации состояния оборудования и технологического процесса.

На таких компьютерах, как правило, в качестве операционной системы (ОС) используют классические ОС (с разделением времени) с дополнительными программными комплексами, адаптирующими их к требованиям реального времени.

Промышленные компьютеры. Состоят из одной платы, на которой размещены процессор, контролер памяти и память различных видов (ОЗУ, ПЗУ, статическое ОЗУ, флэш-память).

Не смотря на наличие контроллеров SCSI (Small Computer System Interface) очень часто СРВ работает без дисковых накопителей. Это связано с тем, что дисковые накопители не отвечают требованиям, предъявляемым к системам реального времени, таким как надежность, устойчивость к вибрациям, габаритам и времени готовности после включения питания.

Плата помещается в специальный корпус, в котором установлении разъемы шины и источник питания. Корпус обеспечивает специальный температурный режим, защиту от пыли и механических повреждений. В этот же корпус вставляются цифроаналоговые и аналогово-цифровые преобразователи, через которых осуществляется ввод/вывод управляющей информации, управление электромоторами и т.п.

Среди промышленных процессоров доминируют процессоры семейств PowerPC (Motorola — IBM), Motorola 68xxx (Motorola). Так же широкую нишу занимают процессоры семейства SPARC (SUN), Intel (Intel), ARM (ARM).

При выборе процессора определяющими факторами являются получение требуемой производительности при наименьшей тактовой частоте, а так же время между переключением задач и реакции на прерывания.

Промышленные компьютеры используются для непосредственного управления промышленным или иным оборудовани-

ем. Они часто не имеют монитора и клавиатуры. Для взаимодействия с ними используются обычные компьютеры, соединенные с ними через порты или Ethernet.

Отметим основные особенности СРВ, диктуемые необходимостью работы на промышленном компьютере:

– Система часто должна работать на бездисковом компьютере и осуществлять начальную загрузку из ПЗУ. В силу этого должны учитываться следующие факторы:
  - критически важным является размер системы;
  - для экономии места в ПЗУ часть системы хранится в сжатом виде и загружается в ОЗУ по мере необходимости;
  - система часто позволяет исполнять код, как в ОЗУ, так и в ПЗУ;
  - при наличии свободного места в ОЗУ система часто копирует код из более медленного ПЗУ в ОЗУ;
  - сама система, как правило, создается на другом компьютере — «обычном» компьютере.
– Система должна по возможности использовать как можно большее число типов процессоров, что дает возможность потребителю выбрать процессор необходимой мощности.
– Система должна по возможности поддерживать более широкий ряд специального оборудования (периферийные контроллеры, таймеры и т.д.).

Критически важным параметром является возможность предсказания времени реакции на прерывания.

В целом ряде задач автоматизации программные комплексы должны работать как составная часть более крупных автоматических систем без непосредственного участия человека. В таких случаях СРВ называют встраиваемыми.

Встраиваемые системы (Embedded system) можно определить как программное и аппаратное обеспечение, составляющее компоненты другой, большей системы и работающее без вмешательства человека [3]. Встраиваемые системы устанавливаются внутрь оборудования, которым они управляют. Для крупного оборудования совпадают с промышленными компьютерами. Для меньшего оборудования представляют собой про-

цессор с сопутствующими элементами, размещенными на одной плате с другими электронными компонентами этого оборудования.

1.3. Организация систем реального времени

1.3.1. Типичное строение систем реального времени

Типичная схема системы реального времени показана на рис. 1.2. Система состоит из трех подсистем: операционной, контролируемой (управляемой) и контролирующей. Между этими подсистемами существуют интерфейсы: приложения и машинный.

Контролируемая подсистема диктует требования в реальном масштабе времени и выдает основные характеристики объекта управления.

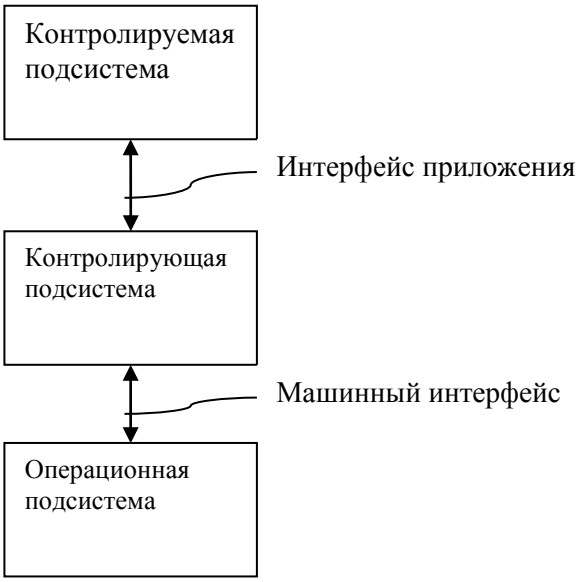

Рис. 1.2. Типичное строение систем реального времени

Контролируемая подсистема представлена задачами (в дальнейшем называемыми прикладными задачами) которые используют оборудование, управляемое подсистемой контроля.

Контролирующая подсистема управляет вычислениями, управляет связью с внешним оборудованием. Контролирующая подсистема должна обеспечивать распределение ресурсов, таких как память, доступ к сети, устройство длительного хранения информации.

Операционная подсистема обеспечивает связь с оператором. Контролирует полную деятельность системы. Данное программное обеспечение носит название **операционной системой реального масштаба времени** (RTOS – real time operating system). Операционные системы реального времени (ОСРВ) — это специальный класс программного обеспечения нижнего уровня, на базе которого разрабатываются системы реального времени.

Интерфейс приложения реализуется с помощью датчиков и исполнительных механизмов.

Машинный интерфейс обеспечивает связь конечных устройств информационной системы с подсистемой визуализации оператора.

1.3.2. Задачи, решаемые в системах реального времени

СРВ представляют собой набор взаимодействующих между собой заданий или задач. Задача является одиночным объектом, управление которым осуществляется оболочкой СРВ. В зависимости от количества задач и от их вида определяется время функционирования СРВ.

Задачи классифицируют по двум категориям:

I Категория — По времени функционирования:

- задачи в ЖРВ (жестком реальном времени);
- задачи в МРВ (мягком реальном времени);
- задачи в «нереальном времени».

II Категория — По типу функционирования:

- периодические задачи;
- апериодические задачи (асинхронные);

– спорадические задачи;
– фоновые задачи;
– аппендикс.

Первая категория задач соответствует определения типов систем реального времени данных в пункте 1.1. (системы жесткого и реального времени). Задачи «нереального времени» — это задачи, для которой нет требований по своевременному выполнению.

Рассмотрим более подробно **задачи второй категории.**

Периодические задачи — это задачи, которые переходят в состояние выполнения через строго заданный период и выполняются каждый цикл функционирования в системе. Например, обработка и контроль сигнала. Для СРВ требуется четкое и своевременное выполнение каждой периодической задачи

Периодическая задача выполняется в строго отведенное ей время, каждый цикл. Запуск периодической задачи может осуществляться несколько раз за цикл. Характеризуется жестким крайним сроком исполнения.

Апериодические задачи — это задачи, имеющие минимальный приоритет в системе и выполняющиеся по событию. Характеризуются наличием мягкого крайнего срока исполнения.

Функционирование таких задач осуществляется только в том случае, если периодические задачи не выполняются. К функциям апериодических задач относятся функции диагностики, выдача справочной информации и сохранение информации на внешнем носителе.

Спорадические задачи — это апериодические задачи с жестким крайним сроком исполнения. Приоритет устанавливается на уровне периодических задач.

Спорадические задачи имеют непредсказуемый характер. Для обработки выделяется отдельная периодическая задача, которая будет контролировать выполнение.

Фоновые задачи — это задачи, для которых предельный срок исполнения не задается, либо устанавливается мягкий крайний срок исполнения. Может исполняться один раз за несколько циклов функционирования системы.

Задачи аппендиксы — это задачи, которые исполняются до старта ОС и имеют приоритет выше, чем сама ОС.

Данные задачи связаны с доступом к аппаратуре, например, установка триггеров, регистров и временных меток.

1.3.3. Архитектура приложений систем реального времени с учетом предсказуемости

В системах реального времени существуют две парадигмы приложений с учетом предсказуемости систем:

– Архитектура приложения, работающего по событию. (Event Type).

– Архитектура приложения, функционирующего по времени. (Time Type).

Event Type. Любая деятельность системы начинается в ответ на возникающее специфическое событие. Вид события определяется самой системой. Предсказуемость достигается следующими способами:

1. Использование стратегии оценки для каждой прикладной задачи (оценивается потребность данной задачи в текущий момент времени). Проверяется несколькими способами:

а) Проверяется загруженность узла.

б) Проверяется время предыдущего запуска данной задачи и анализируется эффект от невыполнения задачи. При отрицательном эффекте задача обязательно должна быть запущена.

2. Оценка потребности ресурсов для данной задачи.

3. Оценка готовности ресурсов для удовлетворения потребностей и задач.

Достоинства: управляемость со стороны системы, независимость от времени (количества тактов).

Недостатки: сложность алгоритма оценки, отсутствие возможности синхронизации событий на разных узлах.

Time Type. Деятельность системы начинается в определенный заданный момент глобально синхронизированного времени. Предсказуемость достигается путём приведения всех задач к периодическим. Для апериодических, спорадических и фоновых задач создаются метазадачи, которые занимаются обработкой соответствующего типа задач.

Достоинства: на всех узлах задачи могут исполняться по синхронизированному времени; таблица задач является фиксированной, для неё можно провести моделирование на возможность функционирования в режиме РВ.

Недостатки: слабая управляемость процесса исполнения задач. Не существует возможности управления последовательностью задач в процессе функционирования системы.

1.3.4. Проектирование систем жесткого реального времени

Важнейшей стадией при разработке любой системы реального времени является создание проекта, который удовлетворяет ряду важных требований. Системы реального времени отличаются от обычных систем обработки данных тем, что к ним применяются некоторые нефункциональные требования (надежность и распределение времени). Как правило, стандартные методы проектирования не дают хороших результатов.

Роль нефункциональных требований в процессе проектирования. В настоящее время все больше заметно, что роль и важность нефункциональных требований в разработке комплексных приложений оценивается неадекватно. Для разработчиков систем, для методов, которые они используют, характерна концентрация в первую очередь на функциональности, и лишь потом, сравнительно поздно, — на нефункциональных требованиях. Хотя есть авторы, которые предполагают, что такой подход неверен при производстве безопасных ответственных систем. Например, часто требования по расчету времени рассматриваются в рамках производительности системы как целого. Отсутствие необходимой производительности часто выливается в какие-либо специальные ее изменения.

Нефункциональные требования включают в себя надежность, своевременность и управление динамическими изменениями (т.е. занесение эволюционных изменений в работающую систему). Эти требования и условия, вносимые средой исполнения, должны приниматься во внимание во время разработки. Во время разработки необходима ранняя привязка программных

функций к компонентам устройств с тем, чтобы можно было проводить анализ распределения времени и надежных характеристик еще не отлаженной системы.

Для того чтобы методы проектирования адекватно учитывали особенности систем жесткого реального времени, они должны поддерживать:

– четкое разделение типов действий/объектов, которые находятся в системах жесткого реального времени (т.е. циклические и единичные действия);
– точное определение требований приложения по распределению времени для каждого объекта;
– определение относительной важности каждого объекта для успешного функционирования приложения;
– точное определение и использование объектов контроля ресурсов;
– переход к наиболее подходящей для планировки и распределения времени программной архитектуре.

Кроме того, методы проектирования должны допускать влияние планировки на проект как можно раньше.

Архитектурный план систем жесткого реального времени. При проектировании систем жесткого реального времени разрабатывают архитектурный план, который включает две фазы:

– логическая архитектура;
– физическая архитектура.

Логическая архитектура включает действия, которые могут быть проделаны независимо от условий, накладываемых средой исполнения, и в первую очередь направлены на удовлетворение функциональных требований.

Физическая архитектура принимает в расчет эти и другие условия и вдобавок охватывает нефункциональные требования. Например, если все объекты построены с учетом худших условий по распределению времени и надежности, то сама система будет удовлетворять требованиям сохранности. Таким образом, физическая архитектура позволяет оценить параметры разработки для достижения компромиссного решения для всех требований задачи.

Когда архитектурные фазы закончены, можно начинать серьезное планирование деталей проекта. Когда это будет сделано, нужно измерить характеристики выполнения кода, чтобы гарантировать, что верхние оценки времени выполнения в самом деле корректны. Если же это не так (что обычно имеет место для новых приложений), фаза физической архитектуры пересматривается и обновляется. Если же все-таки система не реализуется, тогда либо должны быть пересмотрены детали проекта (при небольших отклонениях), или разработчик должен вернуться к фазе логической архитектуры (при серьезных проблемах). Когда измерения кода пройдены, выполняется тестирование приложения. Оно должно включать действительные распределения времени по коду.

Проектирование логической архитектуры. Существует два аспекта любого метода проектирования, которые облегчают создание логической архитектуры жестких систем реального времени. Во-первых, абстракции должна быть дана конкретная поддержка, что, как правило, и нужно проектировщикам систем. Во-вторых, логическая архитектура должна планироваться с тем условием, чтобы возможно было ее анализировать во время проектирования физической архитектуры.

Результатом иерархического разделения в ходе фазы логического планирования является коррекция конечных объектов с полным определением их взаимодействия. Предполагается, что некоторые формы процесса функционального разделения могут приводить к определению новых объектов.

Конечные объекты характеризуются как:

- циклические;
- единичные;
- защищенные;
- пассивные.

Циклические и единичные действия являются обычными в системах реального времени; каждое из них должно содержать поток, который создается во время выполнения. Приоритет потока устанавливается во время планировки в фазе физической архитектуры***. Защищенные объекты*** управляют доступом к

данным, которые доступны нескольким потокам (т.е. циклическим и единичным объектам); в частности, они предусматривают взаимное исключение. В однопроцессорных системах это достигается определением верхнего приоритета для каждого защищенного объекта, который равен максимальному количеству потоков, использующих его. Когда поток обращается к защищенному объекту, он начинает работать с верхним приоритетом и, следовательно, получает исключительный доступ к данным, хранимым в защищенном объекте. Защищенные объекты подобны перехватчикам, в которых вызывающий может быть заблокирован, если условия не позволяют ему продолжить. Это используется для задержки единичных объектов, пока не произойдет освобождающее событие. Последним важным типом объектов является ***пассивный***. Он используется для объекта, который или используется только одним потоком, или может использоваться совместно без ошибок.

Все четыре вышеуказанных типа объектов приемлемы как конечные объекты в жестких системах реального времени. Однако возможно, что система реального времени имеет подсистему, которая не является системой реального времени. Объекты в таких подсистемах являются пассивными или активными. Активные типы объектов могут участвовать в разделении главной системы, но должны быть трансформированы в один из вышеуказанных типов до достижения конечного уровня.

С помощью этих типов конечных объектов могут поддерживаться стандартные конструкции, используемые в жестких системах реального времени:

– ***Периодические действия*** — представляется циклическими объектами.

– ***Единичные действия*** — представляется единичными объектами.

Действия, обусловленные приоритетом, влекут серии вычислений на конечных объектах. Вероятно появление в проектах, которые должны отражать транзакционные сроки.

Процесс планирования логической архитектуры может начаться с разработки активных и пассивных объектов. Процесс разделения приведет к разработке конечных объектов соответ-

ствующего характера. Например, требуемая циклическая транзакция от входа к выходу может быть сначала представлена как единичный активный объект, но потом может быть реализована как циклический объект с последующими сериями единичных объектов, связанных защищенными объектами.

Чтобы иметь возможность анализировать весь проект, необходимо поставить определенные условия:

1. Циклические и единичные объекты не могут выполнять произвольные операции блокировки в других циклических или единичных объектах.
2. Циклические и случайные объекты могут выполнять асинхронную передачу операций управления в другие циклические или единичные объекты.
3. Защищенные объекты не могут выполнять операции блокировки в любых других объектах.

Пункты 1 и 2 требуют, чтобы циклическим и единичным объектам было разрешена только связь через посылку полностью асинхронных сообщений или защищенные объекты.

Любой метод проектирования систем реального времени должен учитывать эти условия в процессе планирования логической архитектуры.

Проектирование физической архитектуры. Основное внимание в физической архитектуре уделяется требованиям к распределению времени. Процесс проектирования должен поддерживать формирование физической архитектуры через следующие функции:

1. Возможность ассоциирования атрибутов распределения времени с объектами.
2. Обеспечение такой внутренней структуры, в которой может быть предпринята планировка конечных объектов.
3. Создание абстракции, с помощью которой проектировщик может контролировать ошибки распределения времени.

Физический план должен быть осуществлен в контексте среды исполнения. Это гарантируется планировкой. Вопросы надежности также должны быть рассмотрены в этой фазе.

Все конечные объекты обладают ассоциированными с ними атрибутами реального времени. Многие атрибуты связаны с отображением на логический план требований распределения

времени (например, срок, важность). Они должны быть установлены до того, как будет производиться планировка.

Каждый циклический или единичный объект имеет некоторое количество временных атрибутов. Например:

1. Период исполнения для каждого циклического объекта.
2. Минимальный интервал проявления для единичного объекта.
3. Сроки для всех циклических и единичных действий.

Различаются две формы сроков. Одна форма применяется прямо к единичному или циклическому действию. Другая применяется к ранее обусловленному действию (транзакции). Сроки для других действий должны извлекаться таким образом, чтобы полная транзакция удовлетворяла ее требованиям распределения времени.

Для планировки нужно знать верхнюю оценку времени исполнения каждого потока и все операции (во всех объектах). После фазы логического планирования они могут быть оценены и им присвоены соответствующие атрибуты. Чем лучше оценки, тем точнее планировка. Хорошие оценки могут быть получены при повторном использовании компонент или из аргументов сравнения (с существующими компонентами других проектов). В процессе детального проектирования и кодирования, а также при прямом использовании измерений во время тестирования, могут быть получены лучшие оценки, которые потребуют перепланировки.

Планировка является составной частью разработки физической архитектуры. Предложенный план должен быть осуществим. Это значит, что все сроки должны гарантироваться при всех предсказуемых обстоятельствах. Реализация этого требует знания скорости процессора, скорости памяти и ее емкости, временных характеристик ядра. Могут быть нужны также временные параметры других устройств. Если мы предполагаем, что среда исполнения поддерживает приоритетную опережающую диспетчеризацию потоков, то в этом случае планировка заключается в определении статических приоритетов потоков, заключенных в циклических и единичных действиях.

Контроль над временными ошибками. Описанная выше планировка может быть эффективна, если оценки/измерения верхней границы времени исполнения точны. В области вре-

менных характеристик определяются две стратегии для ослабления результатов ошибок в компонентах программ:

– Не давать объекту вычислительного времени больше, чем ему нужно.

– Не позволять объектам выполняться по истечении срока.

Вопросы для самопроверки

1. Дайте определение системы реального времени.
2. Что понимают под параллельными системами, и какие типы параллельных систем существуют?
3. Приведите классификацию систем реального времени.
4. Что означают термины система жесткого и система мягкого реального времени?
5. В каких областях применяются системы реального времени?
6. На каких платформах существуют системы реального времени?
7. Какие процессоры доминируют среди «промышленных компьютеров»?
8. Приведите типичную структуру построения системы реального времени.
9. Приведите классификацию задач в системах реального времени.
10. Опишите архитектуры приложений систем реального времени с учетом предсказуемости.
11. Приведите описание процесса проектирования системы соответствующее логической архитектуре.
12. Приведите описание процесса проектирования системы соответствующее физической архитектуре.

2. Автоматизированные системы управления технологическими процессами

2.1. Этапы развития АСУТП

Современная автоматизированная система управления технологическим процессом (АСУТП) представляет собой многоуровневую человеко-машинную систему управления, как правило, удовлетворяющую требованиям, предъявляемым к системам жесткого реального времени. Создание АСУ сложными технологическими процессами осуществляется с использованием автоматических информационных систем сбора данных и вычислительных комплексов, которые постоянно совершенствуются по мере эволюции технических средств и программного обеспечения.

Выделяют три этапа развития АСУТП, обусловленные характером объектов и методов управления, средств автоматизации и других компонентов, составляющих содержание системы управления.

Первый этап — системы автоматического регулирования (САР). Объектами управления на этом этапе являются отдельные параметры, установки, агрегаты; решение задач стабилизации, программного управления, слежения переходит от человека к САР. У человека появляются функции расчета задания и параметры настройки регуляторов.

Второй этап — системы автоматического управления. Объектом управления становится рассредоточенная в пространстве система; с помощью САУ реализуются все более сложные законы управления, решаются задачи оптимального и адаптивного управления, проводится идентификация объекта и состояний системы. Характерной особенностью этого этапа является внедрение систем телемеханики в управление технологическими процессами. Человек все больше отдаляется от объекта управления, между объектом и диспетчером выстраивается целый ряд измерительных систем, исполнительных механизмов,

средств телемеханики, мнемосхем и других средств отображения информации (СОИ).

Третий этап — автоматизированные системы управления технологическими процессами. АСУТП характеризуется внедрением в управление технологическими процессами вычислительной техники. Вначале используется применение микропроцессоров на отдельных фазах управления вычислительных систем; затем активно развиваются человеко-машинные системы управления, инженерной психологией, методы и модели исследования операций и, наконец, диспетчерское управление на основе использования автоматических информационных систем сбора данных и современных вычислительных комплексов.

От этапа к этапу меняются ***функции человека*** (оператора/диспетчера), призванного обеспечить регламентное функционирование технологического процесса. Расширяется круг задач, решаемых на уровне управления; ограниченный прямой необходимостью управления технологическим процессом набор задач пополняется качественно новыми задачами, ранее имеющими вспомогательный характер или относящиеся к другому уровню управления.

Необходимым условием эффективной реализации диспетчерского управления, имеющего ярко выраженный динамический характер, становится работа с информацией, т. е. процессы сбора, передачи, обработки, отображения, представления информации. От диспетчера уже требуется не только профессиональное знание технологического процесса, основ управления им, но и опыт работы в информационных системах, умение принимать решение (в диалоге с ЭВМ) в нештатных и аварийных ситуациях и многое другое. Следует отметить и проблему технологического риска. Технологические процессы в энергетике, нефтегазовой и ряде других отраслей промышленности являются потенциально опасными и при возникновении аварий приводят к человеческим жертвам, а также к значительному материальному и экологическому ущербу.

В результате анализа большинства аварий и происшествий на всех видах транспорта, в промышленности и энергетике были получены интересные данные. В 60-х годах ошибка человека была первоначальной причиной аварий лишь в 20% случаев, то-

гда как к концу 80-х доля "человеческого фактора" стала приближаться к 80 %.

Таким образом, требование повышения надежности систем диспетчерского управления является одной из предпосылок появления нового подхода, реализованного в концепции **SCADA (Supervisory Control And Data Acquisition — диспетчерское управление и сбор данных)** и предопределенного ходом развития систем управления и результатами научно-технического прогресса.

SCADA-система представляет собой пакет программ, предназначенный для разработки и реализации компьютерных рабочих станции операторов в системах автоматизации производства, т. е. программные средства, реализующие основные функции визуализации измеряемой и контролируемой информации, передачи данных и команд системе контроля и управления.

Применение SCADA-технологий позволяет достичь высокого уровня автоматизации в решении задач разработки систем управления, сбора, обработки, передачи, хранения и отображения информации.

Дружественность человеко-машинного интерфейса (HMI/MMI), предоставляемого SCADA-системами, полнота и наглядность представляемой на экране информации, доступность «рычагов» управления, удобство пользования подсказками и справочной системой и т. д. — повышает эффективность взаимодействия диспетчера с системой и сводит к нулю его критические ошибки при управлении.

Следует отметить, что концепция SCADA, основу которой составляет автоматизированная разработка систем управления, позволяет сократить сроки разработки проектов по автоматизации и прямые финансовые затраты на их разработку.

В настоящее время SCADA является основным и наиболее перспективным методом автоматизированного управления сложными динамическими системами (процессами).

Управление технологическими процессами на основе систем SCADA стало осуществляться в передовых западных странах в 80-е годы. Область применения охватывает сложные объекты электро- и водоснабжения, химические, нефтехимические

и нефтеперерабатывающие производства, железнодорожный транспорт, транспорт нефти и газа и др.

В мире насчитывается не один десяток компаний, активно занимающихся разработкой и внедрением SCADA-систем. Каждая SCADA-система — это уникальная инновационная разработка компании и поэтому данные о той или иной системе не столь обширны. В СНГ распространяется более 20-ти разных открытых SCADA-программ, в основном зарубежных и частично отечественных производителей. Некоторые из этих SCADA-программ специализированы на конкретные отрасли производства и задачи, но подавляющее большинство SCADA-программ имеют универсальный характер. Ниже перечисляются отдельные SCADA-программы, используемые на предприятиях СНГ.

Среди SCADA-программ зарубежных производителей, используемых на предприятиях СНГ можно отметить Bridge View и Lookout (National Instuments, США), Cimplicity (GE Fanuc, США), Citect (Ci Technologies, Австралия), Factory Link (U.S.Data Co, США), Genesis (Iconics Co, США), iFIX (Intellution, США), InTouch (Wonderware, США), Maestro NT, SattGraf 5000 и MicroSCADA (ABB, США), RealFlex (BJ Software Systems, США, RSView32 (Rockwell Automation, Великобритания) и другие. Назовем SCADA-программы отечественных производителей: Круг 2000 (Круг), САРГОН (НВТ Автоматика), СКАТ-М (Центрпрограммсистем), RTWin CACSD (SWD Real time System), Trace Mode (AdAstra), Viord micro SCADA (Фиорд), VNS (ИнСАТ).

SCADA-система состоит из инструментального комплекса (средство разработки конкретного программного обеспечения разных серверов и рабочих станций) и исполнительского комплекса (реализация разработанного программного обеспечения в определенной операционной среде). С каждым годом уменьшается число фирм, которые сами разрабатывают для своего промышленного технологического комплекса оригинальные SCADA программы, и все увеличивается число специализированных фирм, которые выпускают открытые SCADA программы, используемые практически в любых промышленно технологических комплексах и с любыми контроллерами.

2.2. Назначение компонентов систем контроля и управления

Специфика каждой конкретной системы диспетчерского управления и сбора данных определяется используемой программно-аппаратной платформой используемой на каждом из двух уровней (нижний и верхний уровень).

Нижний уровень — контроллерный уровень объекта, который включает различные датчики для сбора информации о ходе технологического процесса, электроприводы и исполнительные механизмы для реализации регулирующих и управляющих воздействий. Датчики поставляют информацию локальным программируемым логическим контроллерам (PLC — Programming Logical Controller), которые могут выполнять следующие функции:

– сбор и обработка информации о параметрах технологического процесса;

– управление электроприводами и другими исполнительными механизмами;

– решение задач автоматического логического управления и др.

Как правило, информация в контроллерах предварительно обрабатывается и частично используется на месте. Это существенно снижает требования к пропускной способности каналов связи.

В качестве локальных PLC в системах контроля и управления различными технологическими процессами в настоящее время применяются контроллеры, как отечественных производителей, так и зарубежных. На рынке представлены многие десятки и даже сотни типов контроллеров, способных обрабатывать от нескольких переменных до нескольких сот переменных.

К аппаратно-программным средствам контроллерного уровня управления предъявляются ***жесткие требования по надежности, времени реакции на исполнительные устройства*** и т.д. Программируемые логические контроллеры должны гарантированно откликаться ***на внешние события, поступаю-***

щие от объекта, за время, определенное для каждого события.

Для критичных с этой точки зрения объектов рекомендуется использовать контроллеры с операционными системами реального времени (ОСРВ). ***Контроллеры под управлением ОСРВ функционируют в режиме жесткого реального времени.***

Информация с локальных контроллеров может направляться в сеть диспетчерского пункта непосредственно, а также через контроллеры верхнего уровня. В зависимости от поставленной задачи контроллеры верхнего уровня (концентраторы, интеллектуальные или коммуникационные контроллеры) реализуют различные функции. Некоторые из них перечислены ниже:

– сбор данных с локальных контроллеров;
– обработка данных, включая масштабирование;
– поддержание единого времени в системе;
– синхронизация работы подсистем;
– организация архивов по выбранным параметрам;
– обмен информацией между локальными контроллерами и верхним уровнем;
– работа в автономном режиме при нарушениях связи с верхним уровнем;
– резервирование каналов передачи данных и др.

Верхний уровень — диспетчерский пункт, который включает, прежде всего, одну или несколько станций управления, представляющих собой автоматизированное рабочее место (АРМ) диспетчера/оператора. Здесь же может быть размещен сервер базы данных, рабочие места для специалистов и т. д. Часто в качестве рабочих станций используются ПЭВМ типа IBM PC различных конфигураций.

Станции управления предназначены для отображения хода технологического процесса и оперативного управления. Эти задачи и призваны решать SCADA-системы.

Спектр функциональных возможностей определен самой ролью SCADA в системах управления и реализован практически во всех пакетах. Перечислим часть спектра функциональных возможностей:

– автоматизированная разработка, дающая возможность создания программного обеспечения для системы автоматизации без реального программирования;
– средства исполнения прикладных программ;
– сбор первичной информации от устройств нижнего уровня;
– обработка первичной информации;
– регистрация сигналов тревоги и исторических данных;
– хранение информации с возможностью ее последующей обработки (как правило, реализуется через интерфейсы к наиболее популярным базам данных);
– визуализация информации в виде мнемосхем, графиков и т.п.;
– возможность работы прикладной системы с наборами параметров, рассматриваемых как «единое целое».

Обеспечение взаимодействия SCADA — систем с локальными контроллерами, контроллерами верхнего уровня, офисными и промышленными сетями возложено на, так называемое, ***коммуникационное программное обеспечение***. Это достаточно широкий класс программного обеспечения, выбор которого для конкретной системы управления определяется многими факторами, в том числе и типом применяемых контроллеров, и используемой SCADA — системой.

Большой объем информации, непрерывно поступающий с устройств ввода/вывода систем управления, предопределяет наличие в таких системах баз данных. Основная задача баз данных — своевременно обеспечить пользователя всех уровней управления требуемой информацией. Но если на верхних уровнях АСУ эта задача решена с помощью традиционных баз данныз, то этого не скажешь об уровне АСУ ТП. До недавнего времени регистрация информации в реальном времени решалась на базе программного обеспечения интеллектуальных контроллеров и SCADA — систем. В последнее время появились новые возможности по обеспечению высокоскоростного хранения информации в базах данных, например Интернет.

Перспективные SCADA-программы имеют ***32-х разрядную арифметику и клиент-серверную архитектуру***. При этой архитектуре контроллеры по промышленной сети связаны с сер-

верами, а клиенты (рабочие станции операторов) взаимодействуют по информационной сети с серверами. Такая архитектура для малых систем может быть локальной, тогда и сервер, и клиент размещаются на одном компьютере; а для больших систем — распределенной, тогда клиент и серверы распределены по разным узлам информационной сети. Обычно, в больших системах при наличии многих серверов каждый клиент может информационно взаимодействовать с рядом серверов.

Важно отметить тенденцию включения SCADA-программ в более общий набор взаимосвязанных типовых программных пакетов. Пакеты имеют единые СУБД, как реального времени, так и архивные, и совместно реализуют функции контроля и управления на разных иерархических уровнях производства. Примерный набор таких пакетов включает в себя:

– технологические языки программирования контроллеров (пакет программирования алгоритмов контроля и управления);
– SCADA-программу для оператора;
– SCADA-программу для инженерного персонала, реализующую только функции мониторинга;
– SCADA-программу для диспетчера всего производства, включающую в себя функции планирования и управления материальными и энергетическими потоками;
– комплект программно-логического управления периодическими и полунепрерывными технологическими процессами;
– систему обмена производственной информацией (текущими сигналами датчиков, наблюдаемыми событиями, архивными данными, графическими экранами SCADA-программы) средствами Internet/Intranet с удаленными пультами, разными системами внутри и вне предприятия, руководящим персоналом в любой точке планеты.

Перспективные SCADA-программы обычно состоят из набора самостоятельных программных модулей, каждый из которых выполняет свой комплекс задач и через единые типовые интерфейсы взаимодействует с другими модулями SCADA-программы. Иногда это взаимодействие реализуется через специальное ядро SCADA-программы, но чаще модули взаимодействуют непосредственно, используя типовую технологию

COM/DCOM и объекты ActiveX (подробное их описание приведено в предыдущем разделе). Возможен нижеследующий набор модулей:

– графический векторный редактор с библиотеками графических примитивов и динамическими изображениями типовых производственных объектов;
– серверная станция с СУБД реального времени и архивом;
– модуль обработки событий и тревог;
– генератор отчетов;
– модуль конфигурирования и реализации трендов;
– модуль математических и логических операций (конфигуратор с библиотекой типовых программных модулей контроля и управления);
– модуль статистической обработки данных;
– модуль взаимосвязи в реальном времени между клиентом и сервером;
– модуль обмена данными с приложениями и другими системами и т.д.

Повышение надежности работы SCADA-программы достигается диагностированием неисправностей и резервированием серверов, рабочих станций или отдельных исполняемых ими функций. Диагностируются также обрывы сетей, соединяющих сервер с контроллерами и рабочих станций с сервером.

Горячее резервирование сервера выглядит следующим образом: резервный сервер каждый цикл получает все текущие данные от основного сервера. Но если в очередной цикл данные от него не поступают (неисправность основного сервера), то резервный сервер сам подключается к промышленной сети и работает с нею до тех пор, пока он снова не начнет получать данные от основного сервера.

Повышение надежности решения разных задач в сервере достигается также разделением функций сервера и разделением баз данных на отдельные группы задач:

– сервер работы с текущими сигналами ввода/вывода;
– сервер обработки графической информации;
– сервер поддержки отчетов;

– сервер обслуживания текущих событий и тревог.

Резервирование сетей имеет ряд вариантов:

– возможно полное резервирование всех элементов сетей;

– возможно резервирование только физической среды передачи данных или только аппаратуры сети (сетевых контроллеров и повторителей);

– возможно резервирование связи сервера с контроллерами через дополнительные связи, минуя промышленную сеть, например, связями типа «точка к точке» (point-to-point).

Резервирование рабочих станций или их отдельных функций практически не требует специальных действий. Оно достигается назначением для дублированных рабочих станций одних и тех же уровней доступа к информации и реализацией на них одних и тех же исполнительных комплексов SCADA-программ.

2.3. Функциональные возможности SCADA-систем

Современные SCADA-программы обладают широким набором средств для создания полноценных и удобных для операторов пультов. ***Перечислим основные функции, реализуемые перспективными SCADA-программами***:

– векторная графика с любыми динамическими элементами и широкой палитрой цветов;

– широкий набор графических примитивов и изображений типовых производственных объектов, позволяющий собирать из них различные производственные мнемосхемы и другие графические кадры;

– импорт и экспорт растровых изображений в типовые форматы;

– сохранение и тиражирование созданных мнемосхем, образов и кадров;

– возможности анимации и мультимедиа;

– обработка сообщений и тревог и распределение их по приоритетам;

– автоматический вывод тревог на пейджеры, удаленные рабочие станции, в виде речевых сообщений;
– вывод в окна экрана монитора телеизображений;
– наличие графического конфигуратора с набором типовых программных модулей для вычисления технико-экономических, учетных, статистических показателей контролируемого процесса;
– широкий набор оперативных, архивных и смешанных трендов измеряемых и вычисляемых величин и графиков зависимостей одной измеряемой величины от другой с возможностями их модификаций и изменениями масштаба оператором в реальном времени.

Для эффективного безошибочного внедрения SCADA-программ в них заложены специальные средства тестирования разработанных рабочих станций операторов с возможностью эмуляции конкретного объекта автоматизации.

Безопасная эксплуатация SCADA-программ достигается исключением несанкционированного доступа к ним. Это реализуется путем введения разнообразных по возможностям уровней доступа к информации SCADA-программы и к возможным воздействиям через нее на автоматизируемый объект, а на каждом уровне введением многочисленных паролей, идентифицирующих конкретных пользователей SCADA-программы.

Одной из основных черт современного мира систем автоматизации является их высокая степень интеграции. В любой из них могут быть задействованы объекты управления, исполнительные механизмы, аппаратура, регистрирующая и обрабатывающая информацию, рабочие места операторов, серверы баз данных и т.д. Очевидно, что для эффективного функционирования в этой разнородной среде SCADA-система должна обеспечивать высокий уровень сетевого сервиса. Желательно, чтобы она поддерживала работу в стандартных сетевых средах (***ARCNET, ETHERNET*** и т.д.) с использованием стандартных протоколов (***NETBIOS, TCP/IP*** и др.), а также обеспечивала поддержку наиболее популярных сетевых стандартов из класса промышленных интерфейсов (***PROFIBUS, CANBUS, LON, MODBUS и т.д.***).

Большинство SCADA-систем имеют встроенные языки высокого уровня, VBasic-подобные языки, позволяющие генерировать адекватную реакцию на события, связанные с изменением значения переменной, с выполнением некоторого логического условия, с нажатием комбинации клавиш, а также с выполнением некоторого фрагмента с заданной частотой относительно всего приложения или отдельного окна.

Одной из основных задач систем диспетчерского контроля и управления является обработка информации: сбор, оперативный анализ, хранение, сжатие, пересылка и т. д. Таким образом, в рамках создаваемой системы должна функционировать база данных.

Практически все SCADA-системы, в частности, Genesis, InTouch, Citect, используют ANSI SQL синтаксис, который является независимым от типа базы данных. Таким образом, приложения виртуально изолированы, что позволяет менять базу данных без серьезного изменения самой прикладной задачи, создавать независимые программы для анализа информации, использовать уже наработанное программное обеспечение, ориентированное на обработку данных.

Для специалиста-разработчика системы автоматизации, также как и для специалиста — «технолога», чье рабочее место создается, очень важен графический пользовательский интерфейс. Функционально графические интерфейсы SCADA-систем весьма похожи. В каждой из них существует графический объектно-ориентированный редактор с определенным набором анимационных функций. Используемая векторная графика дает возможность осуществлять широкий набор операций над выбранным объектом, а также быстро обновлять изображение на экране, используя средства анимации.

Крайне важен также вопрос о поддержке в рассматриваемых системах стандартных функций GUI (Graphic Users Interface).

2.4. Контроллеры

Центральное звено систем автоматизации — микропроцессорный контроллер — объединяет под этим названи-

ем ряд классов и типов универсальных микропроцессорных средств, которые удовлетворяют запросам разных категорий заказчиков.

По мощности, косвенно характеризуемой числом обслуживаемых входов/выходов, контроллеры подразделяются на следующие классы:

- класс самых малых контроллеров (десятки входов/выходов);
- класс малых контроллеров (сотни входов/выходов);
- класс больших контроллеров (тысячи входов/выходов).

По области применения из общего множества универсальных контроллеров выделяются несколько подмножеств. Важнейшими из них являются:

- противоаварийные контроллеры — контроллеры повышенной надежности, имеющие определенные сертификаты на работу в цепях аварийной защиты;
- телемеханические контроллеры — контроллеры, оснащенные телемеханическими средствами связи.

По конструктивному исполнению контроллеры подразделяются на встраиваемые в промышленное оборудование (бескорпусное исполнение) и на самостоятельное приборное оформление в виде каркасов, рам, стоек, настенных и напольных шкафов.

Производители контроллеров, работающие на современном уровне, стараются наиболее полно, но без ненужных излишеств удовлетворять разнообразным запросам заказчиков; причем учитывать, что их требования могут при эксплуатации контроллеров со временем изменяться. Для этого применяются следующие технологии построения контроллеров:

- параллельный выпуск серии модификаций контроллеров. Эти серии модификаций имеют одинаковую структуру, однотипные сетевые интерфейсы, одно и то же программное обеспечение и языки программирования. Их отличает друг от друга мощность центрального процессора, частота его работы, наличие разных видов памяти, объем памяти, число и наименование интерфейсов, возможности диагностики и резервирования. Многие производители выпускают до десятка и более вариантов контроллеров одной серии;

– компоновка контроллера по заданным заказчиком требованиям из широкого набора модулей, объединяемых стандартной шиной; так называемая магистрально-модульная архитектура контроллеров.

Последняя технология имеет наибольшие перспективы, она является наиболее гибкой и требует минимум затрат при необходимости модифицировать отдельные характеристики контроллера в процессе его эксплуатации.

На рынке СНГ реализуются контроллеры разных классов, серии, назначений десятков фирм. Ниже перечисляются отдельные фирмы, работающие на этом секторе рынка.

Зарубежные производители контроллеров: *Foxboro, Yokogawa, Fisher-Rosemount, Neles Automation, Honeywell, Moore Products, Rockwell Automation, Koyo Electronics, Schneider Electric, Motorola, Siemens, Triconex, ABB, Omron, GE Fanuc Automation, Toshiba, PEP Modular Computers, FF-Automation, Altersys, Z-World.*

Отечественные производители контроллеров: *Автоваз, Автоматика, ДЭП, Завод электроники и механики, ЗЭИМ Инжиниринг, КОК, КРУГ, НИИтеплоприбор, ПИК ПРОГРЕСС, РИУС, Реалтайм, Системотехника, ТЕКОН, Черноголовка, Эмикон, ЦНИИ Циклон.*

Украинские производители контроллеров: *ИМПУЛЬС, УНИКОНТ, ВЕГА.*

Магистрально-модульная архитектура и ее стандарты. Начало открытым магистрально-модульным системам, имеющим стандартный интерфейс для связи центрального процессора с различными устройствами и блоками ввода/вывода, было положено еще в 1969 году принятием **стандарта CAMAC** (Computer Application for Measurement and Control), получившего широкое распространение в разных странах.[1]

Первой ступенью в системе CAMAC является крейт (каркас), в который вставляют электронные блоки. На задней панели крейта имеется шина обмена. Вся измерительная аппаратура размещается в блоках. В функциональный блок информация по-

[1] Разработка Европейского комитета стандартов ядерной электроники

ступает в виде команд и данных с шины обмена и в виде сигналов от датчиков через переднюю панель. В крейте могут разместиться 23 функциональных блока и специальный блок, называемый контроллером, обеспечивающий связь с каналом обмена ЭВМ. Крейты можно объединять в ветвь, содержащую до 7 крейтов. Контроллеры крейтов подключают к каналу ветви, который через специальный интерфейс, называемый драйвером ветви, соединяется с каналом обмена ЭВМ. Ветвь позволяет разнести крейты и ЭВМ на десятки метров. Для автоматизированных систем, распределенных на большие расстояния, существует последовательный канал САМАС, позволяющий связывать до 62 крейтов. Последовательный канал связан с каналом обмена ЭВМ через специализированный интерфейс, называемый последовательным драйвером.

Эффективность использования систем САМАС обусловлена их гибкостью, возможностью быстрой перестройки и наращивания системы в процессе изменения программы исследований. Причём возможна такая организация работы крейта (и ветви), при которой система обслуживает сразу несколько экспериментов.

Недостаток системы САМАС — малая скорость передачи данных и сложность сведения в систему нескольких процессоров. Разработка и выпуск дешёвых микропроцессоров позволяют создавать многопроцессорные системы.

В конце 80-х годов возникла целая серия открытых магистрально-модульных систем, опирающихся на различные национальные и общественные стандарты. Это системы **VMEbus, STDbus, Mutibus I, Mutibus II, FLTUREbus**.

В 90-е годы наиболее продвинутым по широте распространение (и, в частности, на рынке промышленной автоматики) стал стандарт **VMEbus (Versa Module Eurocard bus) (или VME)** — стандарт на компьютерную шину, первоначально разработанный для семейства микропроцессоров Motorola 68000 (1981 год), и в дальнейшем нашедший применение для множества других приложений.

Шина VMEbus принята в качестве стандарта рядом международных и региональных организаций: IES (стандарт 821)

ANSI, IEEE (стандарт 1014) и др. Более 300 фирм в мире выпускают модули в данном стандарте **VMEbus.**

Основные особенности VMEbus:

– шина VME объединяет модули, размещенные в одном каркасе;

– в каждом каркасе может размещаться до 21 модуля VMEbus. Конструктивное исполнение каркасов может быть любым (стойка, рама, башня, настольное исполнение, шкаф);

– модули, поддерживающие стандарт, процессорно и технологически независимы;

– используются 8/16/32/64-х разрядные архитектуры;

– скорость шины при передаче 64-х разрядных блоков до 130 Мбит/с;

– конструктивно стандарт VMEbus опирается на широко распространенный механический стандарт «Евромеханика». В системе может использоваться любое число каркасов стандарта «Евромеханика», в которых находятся модули VME с объединяющей их магистралью VME и источником питания;

– в одном каркасе может работать несколько различных процессоров, образуя многопроцессорные системы;

– при создании систем, состоящих из многих каркасов VMEbus, для объединения этих каркасов (удлинения шины VME) используется стандартная шина VICbus (VME Interconnect bus). Эта шина позволяет объединить до 15 каркасов на расстоянии 100 м. Скорость передачи данных на VICbus — до 10 Мбайт/с. При этом процессорный модуль может быть только один в любом из каркасов, а остальные каркасы могут содержать пассивные модули ввода/вывода.

Средства VMEbus поддерживают практически все распространенные программные продукты: операционные системы, языки программирования, базы данных, сетевые интерфейсы и т. д.

Номенклатура плат модулей VMEbus составляет более 3000 наименований: центральные процессоры разных типов, архитектуры и разных производителей; сетевые контроллеры практически ко всем распространенным информационным и промышленным сетям; блоки памяти всевозможных видов и

объемов; различные виды модулей вводов/выводов; одно- и многоканальные аналого-цифровые и цифро-аналоговые преобразователи, адаптеры шин персональных компьютеров и т. д.

Органичным дополнением к шине VMEbus является стандартная локальная шина PCI, используемая для внутриплатных соединений. Она позволяет использовать расширяющуюся номенклатуру микроэлектронных внутриплатных PCI-компонентов, повышая производительность и гибкость создаваемых VME средств. Электрическая спецификация PCI в промышленном формате «Евромеханика» носит название **Compact PCI** (Peripheral component interconnect **—** взаимосвязь периферийных компонентов).

CompactPCI — системная шина, широко используемая в промышленной автоматике. Электрически шина отличается от PCI стандарта 2.2, тем что позволяет подключить большее число устройств. Но в целом совместима и обычно использует тот же набор микросхем. Физически разъём выполнен по-другому и позволяет использовать «горячее подключение» плат — то есть, устанавливать и извлекать плату не прерывая работоспособности компьютера. Изделия CompactPCI широко используется в телекомуникациях. Основной конкурент — шины VME, VME32, VME64, являющиеся фактическим стандартом в военной технике НАТО. Широкое применение сдерживалось высокой ценой изделий по сравнению с VME. На настоящий момент базовая шина CompactPCI фактически устарела по пропускной способности. Но разработаны расширения, такие как **PCI-Express** (2002 год). Стандарт CompactPCI позволяет добавить дополнительные сигналы, для передачи данных помимо шины PCI. На основе этих расширений были созданы новые стандарты, такие как **CompactPCI 64** или **PXI**.

В отличие от шины PCI, использовавшей для передачи данных общую шину, PCI Express, в общем случае, является пакетной сетью с топологией типа звезда, устройства PCI Express взаимодействуют между собой через среду, образованную коммутаторами, при этом каждое устройство напрямую связано соединением типа точка-точка с коммутатором.

Кроме того, шиной PCI Express поддерживается:

– горячая замена карт;

– гарантированная полоса пропускания;
– управление энергопотреблением;
– контроль целостности передаваемых данных.

Шина PCI Express нацелена на использование только в качестве локальной шины. Так как программная модель PCI Express во многом унаследована от PCI, то существующие системы и контроллеры могут быть доработаны для использования шины PCI Express заменой только физического уровня, без доработки программного обеспечения. Высокая пиковая производительность шины PCI Express позволяет использовать её вместо шин AGP и тем более PCI и PCI-X (рис. 2.1).

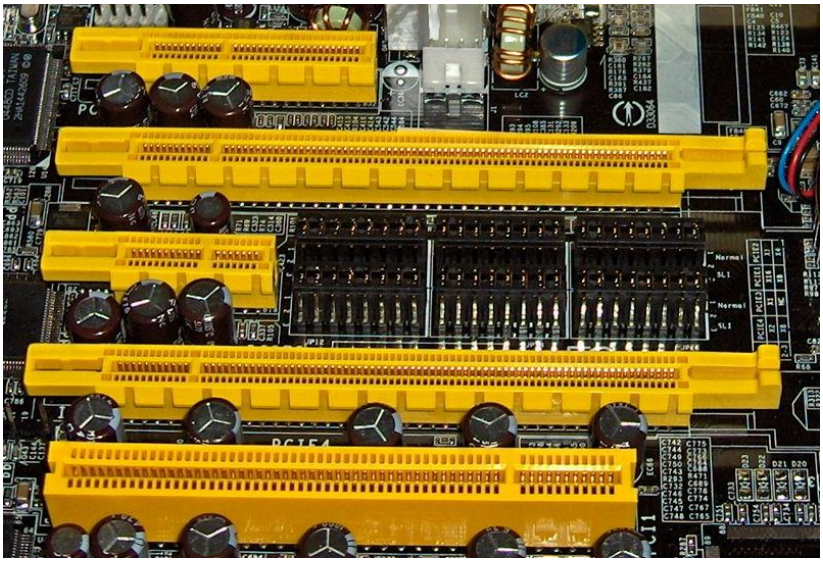

Рис. 2.1. На фотографии 4 слота PCI Express: x4, x16, x1, опять x16, внизу стандартный 32-разрядный слот PCI, на материнской плате DFI LanParty nForce4 SLI-DR

В 2007 году группа PCI-SIG выпустила спецификацию **PCI Express 2.0.** Основные нововведения в PCI Express 2.0:

– Увеличенная пропускная способность — спецификация PCI Express 2.0 определяет максимальную пропускную способность

одного соединения как 5 Гбит/с, при этом сохранена совместимость с PCI Express 1.1 таким образом, что плата расширения, поддерживающая стандарт PCIE 1.1 может работать, будучи установленной в слот PCIE 2.0. Внесены усовершенствования в протокол передачи между устройствами и программную модель:

– Динамическое управление скоростью — для управления скоростью работы связи.
– Оповещение о пропускной способности — для оповещения программного обеспечения (операционной системы, драйверов устройств и т. п.) об изменениях скорости и ширины шины.
– Расширения структуры возможностей — расширение управляющих регистров для лучшего управления устройствами, слотами и интерконнектом.
– Службы управления доступом — опциональные возможности управления транзакциями точка-точка.
– Управление таймаутом выполнения.
– Сброс на уровне функций — опциональный механизм для сброса функций внутри устройства (PCI device).
– Переопределение предела по мощности — для переопределения лимита мощности слота при присоединении устройств, потребляющих большую мощность.

Идёт работа над **PCI-Express 3.0**. Он будет обладать пропускной способностью в 8 Гбит/с. Но, несмотря на это, его реальная пропускная способность всё равно сохранит традицию и будет увеличена вдвое по сравнению со стандартом PCI Express 2.0 благодаря более агрессивной системе связи, однако, совместимость с предыдущими версиями PCI Express сохранится. Стандарт PCI-Express 3.0 будет опубликован не ранее второго квартала 2010 года, а первые продукты на основе нового интерфейса появятся в 2011 году. Выход стандарта задерживается из-за некоторых проблем с совместимостью, так как предыдущие стандарты использовали другие системы связи.

Расширение VME на средства измерительной техники (в частности, на блоки ввода/вывода) называется VXIbus (VMEbus eXtention for Instruments) и стандартизировано в 1991 г. (стандарт IEEE 1155).

VXIbus имеет следующие отличия от VMEbus:

– наличие менеджера ресурсов;
– наличие дополнительной локальной 32-х битной шины;
– наличие дополнительных аналоговой шины и шины идентификации.

Почти все модули VXIbus могут использоваться в VMEbus структурах и все модули VMEbus могут использоваться в VXIbus структурах, но должны поддерживаться стандартными для VXIbus программными драйверами.

В номенклатуру модулей VXIbus входят центральные процессоры, сетевые контроллеры, разные виды памяти, генераторы импульсов и функциональные генераторы, счетчики, таймеры, измерители электрических параметров, аналоговые и цифровые вводы/выводы сигналов разных уровней, аналого-цифровые и цифро-аналоговые преобразователи.

Программным обеспечением являются все открытые операционные системы реального времени. Для программирования модулей VXIbus используется специальный инструментальный язык SCPI (Standart Commands for Programmable Instruments), который реализуется в модулях с помощью имеющихся компиляторов и интерпретаторов с этого языка.

В связи с развитием микроэлектроники все большее число операций может быть реализовано в кристалле и все меньше по размерам могут быть платы, реализующие функционально законченные модули. Это привело к появлению дополнительных малых по размерам, так называемых **«мезонинных» плат**, которые устанавливаются на основной плате (плате-носителе), расширяя е функции. Мезонинные платы представляют собою уровень модульности более низкий, чем уровень модульности плат магистрально-модульной архитектуры (например, VME). Мезонинные модуль имеет небольшие габариты, примерно 45х99 мм (размер кредитной карты) и два разъема: разъем стандартизованной локально шины платы-носителя и разъем, атрибуты которого определяются реализуемой в модуле функцией. Типовая плата-носитель выполненная в стандарте «Евромеханика» формата 6U (размер 233х16 мм), несет на себе 4 мезонинных модуля.

Благодаря стандартизации мезонинный модуль может работать на разных платах-носителях и не зависит от ее устройства свойств.

На сегодня большое распространение получили две стандартные технологии мезонинных модулей: **IndustryPack** и **MODPACK**.

Основные технические характеристики и номенклатуры мезонинных модулей IndustryPack (мезонинные модули MODPACK имеют схожие показатели) приведем ниже.

Технические характеристики:

– скорость до 64 Мбайт/с при тактовой частоте 32 Мгц,
– разрядность данных 16 и 32 бита,
– память до 8-ми Мбайт,
– имеется возможность автоконфигурирования и идентификации,
– работоспособность в диапазоне температур -40 — +850С.

Основная номенклатура (общее число типов модулей достигает 240; ниже перечислены некоторые из них, наиболее существенные для потребителей):

– модули аналогового ввода/вывода на разные типовые сигналы датчиков и исполнительных механизмов, до 16 каналов разрядностью 10-16 бит, с различными видами гальванической развязки;
– модули цифрового ввода/вывода на разные токи и напряжения, с числом каналов от 8 до 48, с различными видами гальванической развязки;
– аналого-цифровые и цифро-аналоговые преобразователи с различными видами гальванической развязки, со временем преобразования сигнала от единиц мкс до сотен мс;
– модули интерфейсов: RS-232, RS-422, RS-485 и др., одно- и многоканальные, со скоростями от 50 бит/с до 10 Мбит/с;
– модули сетевых контроллеров, обслуживающие разные сети: Ethernet, ARCNET, FDDE, PROF1BUS, Bitbus, CANbus и др.;
– модули разных видов памяти разного объема;
– счетчики и таймеры;
– модули обработки изображений (например, от видеокамер) и многие др.

В дополнение к этому ряд фирм стали выпускать типовые мезонинные модули с колодками, в которые можно вставить микросхемы разных типов. Например, в мезонинные модули памяти можно вставить микросхемы памяти разных видов и объемов, а в мезонинные модули ввода/вывода можно установить определенное число мезонинных (съемных) каналов ввода/вывода; причем каждый такой канал может быть любого типа и на любые нужные параметры.

2.5. Технологические языки программирования контроллеров по стандарту IEC 1131.3

Прикладное программное обеспечение контроллеров разрабатывается с помощью специальных технологических языков, большей частью рассчитанных не на квалифицированных программистов, а на специалистов по автоматизации. Наиболее простой и распространенный язык программирования цепей контроля и управления — это набор программных модулей типовых алгоритмов обработки измерительной информации, регулирования, блокировочных зависимостей и графический конфигуратор, который собирает эти модули в заданные цепи контроля и управления. Первоначально каждый производитель промышленных технологических комплексов разрабатывал свои языки программирования контроллеров. В последние годы ситуация начинает изменяться. К середине 90-х годов технологические языки программирования контроллеров были стандартизированы — **стандарт IEC 1131.3**. Этот стандарт определяет структуру пяти технологических языков:

– ***LD — язык лестничных диаграмм***. Графический традиционный язык релейных блокировок, в котором разработчик изображает необходимые релейные схемы;

– ***FBD — язык функциональных блоковых диаграмм***. Графический конфигуратор с набором типовых программных модулей;

– ***SFC — язык последовательных функциональных схем***. Язык, близкий к традиционному программированию, предна-

значен для реализации алгоритмов последовательного управления. Элементы языка — процедуры и транзакции используются для определения порядка операций, написанных на любом языке стандарта;

– ***ST — язык структурированного текста***. Язык типа Pascal, поддерживающий структурное программирование. Он может использоваться для программирования процедур и переходов в языке SFC и дополнять другие языки стандарта;

– ***IL — язык инструкций***. Язык низкого уровня типа Ассемблера, но без ориентации на конкретную микропроцессорную архитектуру. С его помощью можно создавать быстродействующие программные модули.

Два первых графических языка являются основными, а остальные языки служат дополнениями к ним.

В стандарте описываются спецификации механизмов, посредством которых пользователи могут определять новые типы данных, функции и функциональные блоки; то есть стандарт в этом плане является саморасширяющемся. При необходимости многократно применять одну и ту же последовательность функций, можно выделить ее в отдельный функциональный блок, поместить в библиотеку и устанавливать в любые программы управления. Все языки стандарта можно комбинировать; можно также включать в программу фрагменты, написанные на традиционных языках.

Сертификация конкретных языков на соответствие стандарту осуществляется организацией PLCopen, имеющей свои отделения в разных странах. Сформулировано ***три уровня совместимости конкретных языков со стандартом***:

– базовый уровень, когда язык соответствует некоторому подмножеству стандарта IEC 1131.3 (начальный уровень). На этом уровне проверяются типы переменных и языковые конструкции;

– уровень переносимости функций, когда существует формат файла обмена функциональных блоков;

– уровень совместимости и приложений, когда совместимость реализована на уровне приложений и возможен перенос завер-

шенных приложений. Данный уровень спецификации находится в разработке.

Более десятка фирм, специализирующихся на программных продуктах, выпускают сейчас технологические языки по этому стандарту, ориентированные на работу под определенными типовыми операционными системами (т. е. открытые технологические языки), что позволяет разработчикам промышленных технологических комплексов использовать их в своих комплексах. Практически большинство контроллеров, выпускаемых в последние годы, оснащаются тем или иным числом технологических языков, соответствующих этому стандарту. Эти языки либо разработаны самими разработчиками контроллеров, либо закуплены последними у фирм, специализирующихся на программных продуктах.

Перспективные контроллеры в части их прикладного программного обеспечения выделяются следующими факторами:

– они обеспечены технологическими языками по стандарту IEC 1131.3;

– обязательный набор таких языков — языки типа LD и FBD. Желательный набор — все пять языков стандарта;

– эти языки сертифицированы на соответствие стандарту желательно по уровню переносимости функций;

– к языку FBD (функциональных блоковых диаграмм) прилагается обширный (порядка сотен единиц) набор программных модулей типовых алгоритмов контроля и управления;

– в этот набор, кроме обычных простейших функций, входят продвинутые типовые модули управления, повышающие эффективность автоматизации: модуль самонастройки регуляторов, адаптивные регуляторы, регуляторы на нечеткой логике, нейрорегуляторы и т. д.

Открытость контроллеров определяется наличием у них интерфейсов к типовым промышленным и полевым сетям, а также наличием портов последовательной связи с внешними устройствами. Чем большее число таких интерфейсов и стандартных портов имеет контроллер, тем более открытым он является. Достаточно полным открытым набором является наличие протоколов, связывающих контроллер с сетями

PROFIBUS-DP, HART, Fieldbus Ml, Modbus, Ethernet, с сетями Alien Bradley; портов последовательной передачи данных RS-232, RS-422, RS-485; модемов радио и телефонных каналов.

Следует отметить, что сейчас идеология открытости средств автоматизации начинает воплощаться в контроллерах еще одним путем: некоторые фирмы начинают выпускать контроллеры, которые позволяют любым пользователям через сеть собирать информацию с них простейшим гипертекстовым общением (аналогично работе пользователя в Internet).

Вопросы для самопроверки

1. Опишите этапы развития АСУТП.
2. Что представляет собой SCADA-система?
3. Приведите названия популярных зарубежных и отечественных SCADA-программ.
4. Опишите уровни программно-аппаратных платформ системы диспетчерского управления и сбора данных.
5. Опишите, за счет чего достигается повышение надежности работы SCADA-программ.
6. Какие основные функции, реализуются перспективными SCADA-программами?
7. Классифицируйте контроллеры по мощности.
8. Классифицируйте контроллеры по области применения.
9. Классифицируйте контроллеры по конструктивному исполнению.
10. Приведите стандарты магистрально-модульной архитектуры.
11. Опишите спецификации PCI-Express.
12. Опишите стандарт VME.
13. Что представляют собой технологии мезонинных модулей?
14. Для чего используется стандарт IEC 1131.3?
15. Какими факторами выделяются перспективные контроллеры в части их прикладного программного обеспечения?

3. Организация операционных систем реального времени

3.1. Функциональные требования ОСРВ

3.1.1. Основные понятия, используемые при формировании функциональных требований к ОСРВ

Расширение области применения СРВ привело к повышению требований к этим системам. В настоящее время обязательным условием, предъявляемым к ОС, претендующей на применение в задачах реального времени, является реализация в ней ***механизмов многозадачности***. Та же тенденция присутствует и в ОС общего назначения. Но для СРВ к реализации механизмов многозадачности предъявляется ряд дополнительных, по сравнению с системами общего назначения, требований. Определяются эти требования обязательным свойством СРВ — предсказуемостью.

Многозадачность подразумевает параллельное выполнение нескольких действий, однако практическая реализация параллельной работы упирается в проблему совместного использования ресурсов вычислительной системы. И главным ресурсом, распределение которого между несколькими задачами называется ***диспетчеризацией*** (scheduling), является процессор. Поэтому в однопроцессорной системе по-настоящему параллельное выполнение нескольких задач невозможно. Существует достаточно большое количество различных методов диспетчеризации, и основные среди них будут рассмотрены далее.

В многопроцессорных системах проблема ***разделения ресурсов*** также является актуальной, поскольку несколько процессоров вынуждены разделять между собой одну общую шину. Поэтому при построении СРВ, нуждающейся в одновременном решении нескольких задач, применяют группы вычислительных комплексов, объединенных общим управлением. Возможность работы с несколькими процессорами в пределах одного вычислительного комплекса и максимально прозрачное взаимодей-

ствие между несколькими вычислительными комплексами в пределах, скажем, локальной сети, является важной чертой ОСРВ, значительно расширяющей возможности ее применения.

Под понятием задачи в терминах ОС и программных комплексов могут пониматься две разные вещи: ***процессы и потоки***. Процесс является более масштабным представлением задачи, поскольку обозначает независимый модуль программы или весь исполняемый файл целиком с его адресным пространством, состоянием регистров процессора, счетчиком команд, кодом процедур и функций. Поток же является составной частью процесса и обозначает последовательность исполняемого кода. Каждый процесс содержит как минимум один поток, при этом максимальное количество потоков в пределах одного процесса в большинстве ОС ограничено только объемом оперативной памяти вычислительного комплекса. Потоки, принадлежащие одному процессу, разделяют его адресное пространство, поэтому они могут легко обмениваться данными, а время переключения между такими потоками (то есть время, за которое процессор переходит от выполнения команд одного потока к выполнению команд другого) оказывается значительно меньшим, чем время переключение между процессами. В связи с этим в задачах реального времени параллельно выполняемые задачи стараются максимально компоновать в виде потоков, исполняющихся в пределах одного процесса.

Каждый поток имеет важное свойство, на основании которого ОС принимает решение о том, когда предоставить ему время процессора. Это свойство называется ***приоритетом потока*** и выражается целочисленным значением. Количество приоритетов (или уровней приоритетов) определяется функциональными возможностями ОС, при этом самое низкое значение (0) закрепляется за потоком, который предназначен для корректной работы системы, когда ей "ничего не надо делать".

Каждому потоку назначается свой приоритет. Планировщик выбирает поток для выполнения в соответствии с приоритетом каждого потока, находящегося в состоянии готовности, то есть способного использовать процессор. Таким образом выбирается поток с наивысшим приоритетом. На рис. 3.1. схематически изображена очередь из шести потоков (A-F), находящихся в

состоянии готовности. Все остальные потоки (G-Z) блокированы.

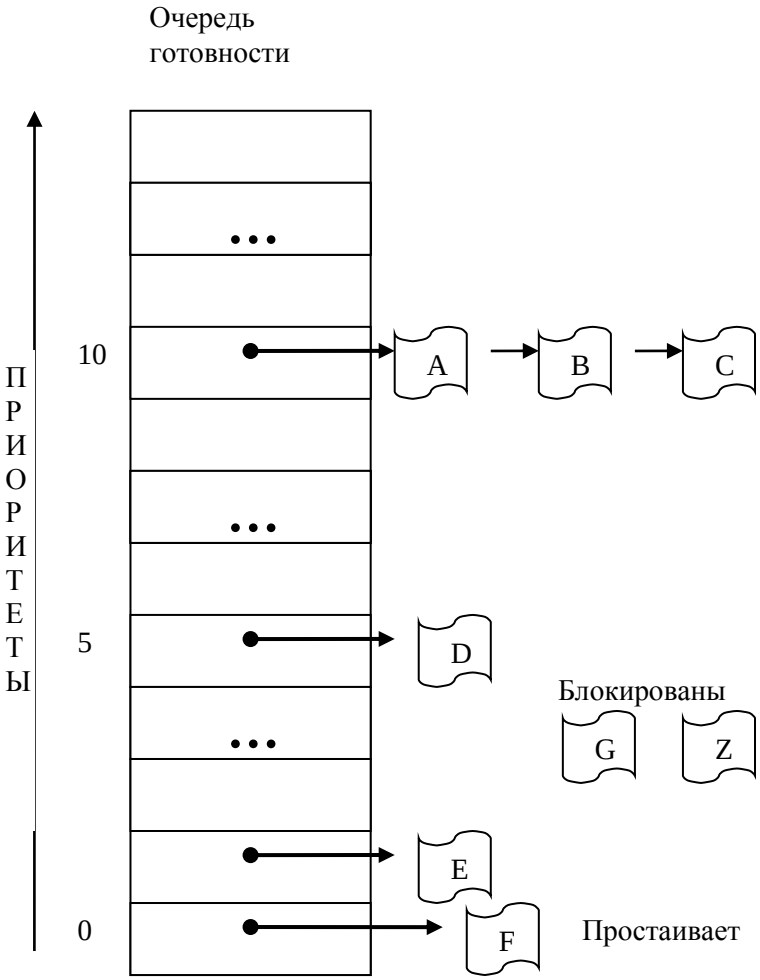

Рис. 3.1. Очередь готовности

Анализ ОС общего назначения, а также, например, ОС реального времени, показывает, что процесс может находиться в

одном из двух основных состояний: **активном или пассивном**. ***В активном*** состоянии процесс может участвовать в конкуренции за использование ресурсов вычислительной системы, а в пассивном — не участвует, хотя в системе и имеется информация об его существовании, что сопряжено с предоставлением ему оперативной и/или внешней памяти. Активный процесс может находиться в одном из следующих состояний [1]:

– ***выполнение:*** затребованные процессом ресурсы выделены. В этом состоянии в каждый момент времени может находиться только один процесс, если речь идет об однопроцессорной вычислительной системе;

– ***готовность к выполнению:*** ресурсы могут быть предоставлены, тогда процесс перейдет в состояние выполнения;

– ***блокирование или ожидание:*** затребованные ресурсы не могут быть предоставлены, или не завершена операция ввода/вывода.

Далее рассматриваются функциональные требования, предъявляемые на данном этапе к ОС, применяющимся в СРВ.

3.1.2. Диспетчеризация потоков

Методы диспетчеризации, т.е. предоставления разным потокам доступа к процессору, в общем случае могут быть разделены на две группы. К первой относятся случаи, когда все потоки, которые разделяют процессор, имеют одинаковый приоритет, т.е. их важность с точки зрения системы одинакова:

– ***FIFO (First In First Out)***. Первой выполняется задача, первой вошедшая в очередь, при этом она выполняется до тех пор, пока не закончит свою работу или не будет заблокирована в ожидании освобождения некоторого ресурса или события. После этого управление передается следующей в очереди задаче (рис.3.2).

– ***Карусельная многозадачность*** (round robin). При этом методе диспетчеризации в системе задается специализированная константа, определяющая продолжительность непрерывного выполнения потока, так называемый квант времени выполнения — time slice (рис.3.3).

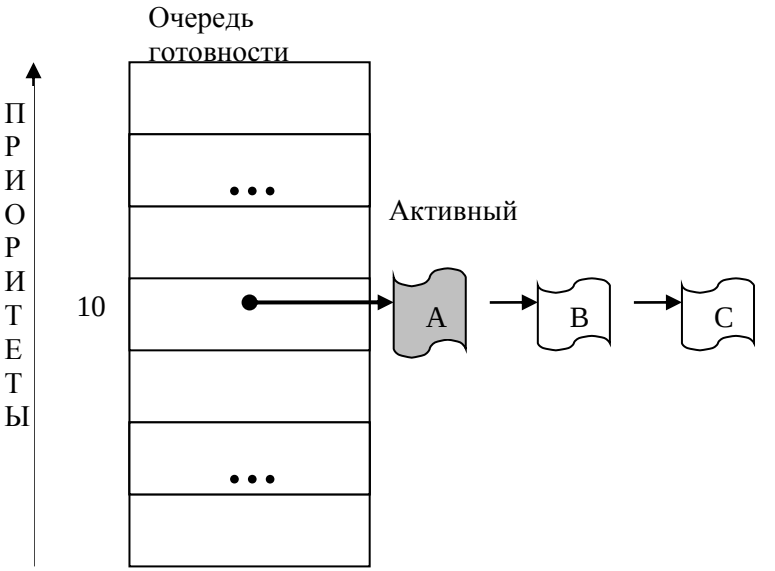

Рис. 3.2. FIFO-планирование

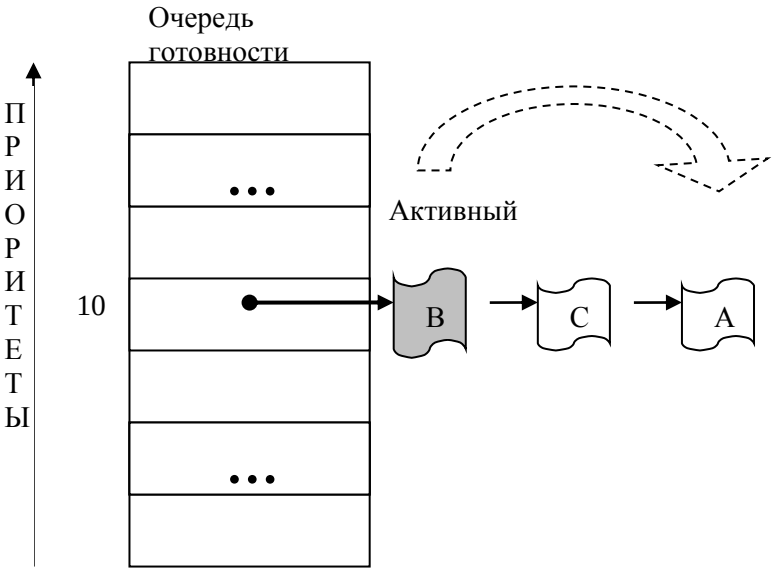

Рис. 3.3. Циклическое планирование

Таким образом, выполнение потока может быть прервано либо окончанием его работы, либо блокированием в ожидании ресурса или события, либо завершением кванта времени (того самого time slice). После этого управление передается следующему в очередности потоку. По окончании времени последнего потока управление передается первому потоку, находящемуся в состоянии готовности. В результате, выполнение каждого потока разбито на последовательность временных циклов.

Появление второй группы методов диспетчеризации связано с необходимостью распределения времени процессора между потоками, имеющими разную важность, т.е. разный приоритет. В таких случаях для потоков с равным приоритетом используется один из указанных выше методов диспетчеризации, а передача управления между потоками с разным приоритетом осуществляется одним из следующих методов:

– В наиболее простом случае если в состояние готовности переходят два потока с разными приоритетами, то процессорное время передается тому, у которого более высокий приоритет. Данный метод называется ***приоритетной многозадачностью*** (рис. 3.4), но его использование в таком виде связано с рядом сложностей. При наличии в системе одной группы потоков с одним приоритетом и другой группы с другим, более низким приоритетом, при карусельной диспетчеризации каждой группы в системе с приоритетной многозадачностью потоки низкоприоритетной группы могут вообще не получить доступа к процессору.

– Одним из решений проблем приоритетной многозадачности стала так называемая ***адаптивная многозадачность*** (рис. 3.5), широко применяющаяся в интерфейсных системах. Суть метода заключается в том, что приоритет потока, не выполняющегося какой-то период времени, повышается на единицу. Восстановление исходного приоритета происходит после выполнения потока в течение одного кванта времени или при блокировке потока. Таким образом, при карусельной многозадачности, очередь (или "карусель") более приоритетных потоков не может полностью заблокировать выполнение очереди менее приоритетных потоков.

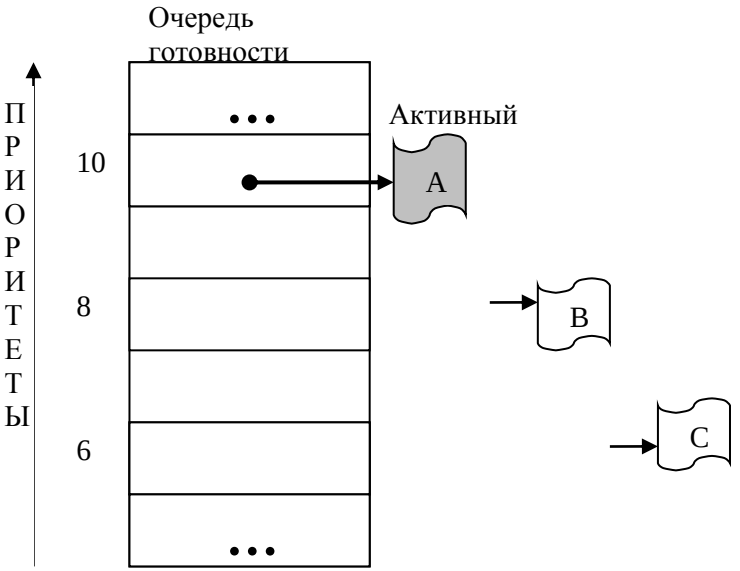

Рис. 3.4. Приоритетное планирование

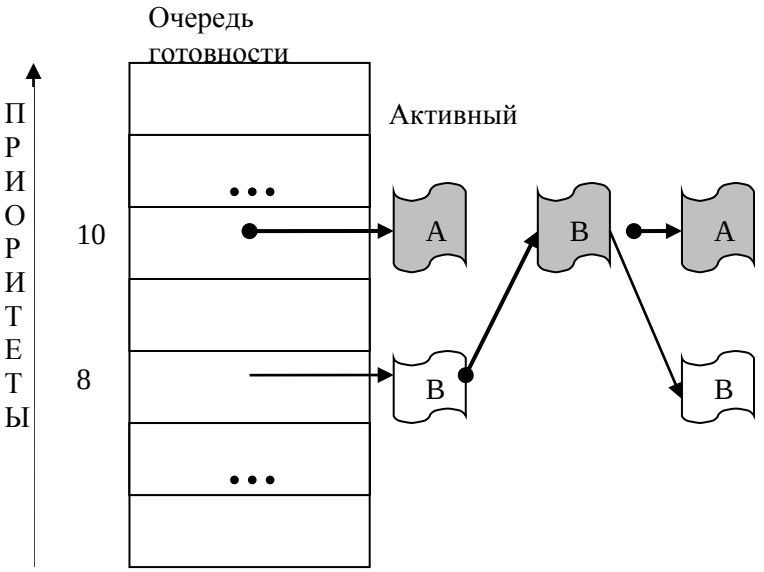

Рис. 3.5. Адаптивное планирование

– В задачах реального времени предъявляются специфические требования к методам диспетчеризации, поскольку передача управления потоку должна определяться критическим сроком его обслуживания (т.н. deadline-driven scheduling). В наибольшей степени этому требованию соответствует ***вытесняющая приоритетная многозадачность***. Суть этого метода заключается в том, что как только поток с более высоким, чем у активного потока, приоритетом переходит в состояние готовности, активный поток вытесняется (т.е. из активного состояния принудительно переходит в состояние готовности) и управление передается более приоритетному потоку.

На практике широко применяются как комбинации описанных методов, так и различные их модификации. В СРВ, в контексте задачи диспетчеризации нескольких потоков с разным приоритетом, очень важной является проблема распределения приоритетов таким образом, чтобы каждый поток уложился в свой критический срок обслуживания. Если все потоки системы укладываются в свои критические сроки обслуживания, то говорят, что система диспетчируема (schedulable).

Для СРВ, применяющихся в обработке периодических событий, в 1970 году Лиу и Лейленд предложили математический аппарат, позволяющий определить, является ли система диспетчируемой. Этот аппарат называется "Частотно монотонный анализ" (ЧМА) (Rate Monotonic Analyzing). Эффективность данного математического аппарата привела к тому, что ЧМА был принят в качестве стандарта такими организациями как USA Department of Defense, Boeing, General Dynamics, Magnavox. Mitre, NASA, Panamax, и др. Среди организаций, установивших ЧМА в качестве стандартного средства анализа и разработки систем жесткого реального времени можно также отметить IBM Federal Sector Division, US Navy и European Space Agency.

Подобная позиция ведущих производителей привела к тому, что разработчикам ОСРВ пришлось учитывать требования по применению ЧМА при разработке своих систем. Возможность применения ЧМА ограничена рядом условий, первым из которых является диспетчеризация потоков методом вытесняющей приоритетной многозадачности.

На основании всего выше сказанного можно сформировать первое требование к ОСРВ: ***ОСРВ должна реализовывать возможность многозадачности, причем с поддержкой вытесняющей приоритетной методики диспетчеризации.***

3.1.3. Уровни приоритетов

Как уже говорилось, для организации параллельного выполнения нескольких потоков зачастую необходимо разделение этих потоков по степени важности (или критическому сроку обслуживания). Среди совокупности параллельно выполняющихся задач выделяются потоки жесткого реального времени, потоки мягкого реального времени и потоки, не критичные ко времени обслуживания. Каждая из указанных групп должна иметь свой уровень приоритетов, к тому же потоки жесткого реального времени, в ряде случаев, должны иметь индивидуальные значения приоритетов. Практический опыт разработки систем реального времени говорит, что увеличение количества разно приоритетных потоков приводит к непрозрачности и непредсказуемости системы. Однако не всегда это выглядит именно так.

На сегодняшний день существует ряд инструментов математического анализа, позволяющих распределить приоритеты между несколькими потоками таким образом, чтобы они гарантировано выполняли свои критические сроки обслуживания. Если же для данного набора потоков реализовать это невозможно, то результаты математического анализа покажут, какие именно потоки имеют критическое отношение срока обслуживания ко времени выполнения.

Упомянутый ранее аппарат ЧМА позволяет провести такое исследование для случая периодических критических времен обслуживания. Однако для его применения анализируемые потоки должны иметь уникальные значения приоритетов, определяемые периодом каждого потока. В связи с этим требованием разработчики ОСРВ закладывают в своих системах достаточно большое количество приоритетов. Для примера в QNX 6.x их 64, а в Windows CE и VxWorks — 256.

Таким образом, можно сформировать второе функциональное требование ОСРВ: ***ОС должна иметь достаточно***

большое (определяется масштабом задачи) количество приоритетов. Рекомендуемым значением является 128 уровней. Естественно, что в прикладных задачах необходимо крайне осторожно использовать потоки с разными приоритетами и, по возможности, стремиться к минимизации их количества. Большое количество потоков с разным приоритетом может привести не только к потере предсказуемости системы, но и к проблемам синхронизации на доступе к разделяемым ресурсам.

3.1.4. Механизмы синхронизации

Помимо процессорного времени разные потоки могут иметь и другие ресурсы, которые им приходится разделять между собой. Это могут быть переменные в памяти, буферы устройств и т.д. Для защиты от искажения, вызванного одновременным редактированием одних и тех же данных разными потоками, используются специфические переменные, называемые объектами синхронизации. К таким объектам относятся мьютексы, семафоры, события и т.д.

Третьим функциональным требованием к ОСРВ является ***наличие в ОС механизмов синхронизации доступа к разделяемым ресурсам***. В принципе механизмы синхронизации присутствуют в любых многозадачных системах, поскольку без них нельзя обеспечить корректную работу нескольких потоков с одним ресурсом (например, буфером устройства или некоторой общей переменной). Однако в задачах реального времени к объектам синхронизации предъявляются специфические требования. Связано это с тем, что именно на объектах синхронизации возможны значительные задержки выполнения потоков, поскольку назначением этих объектов является фактически блокирование доступа к некоторому разделяемому ресурсу. Одной из наиболее серьезных проблем, возможных при блокировании ресурса, является ***инверсия приоритетов***.

3.1.5. Защита от инверсии приоритетов

Итак, проблема инверсии приоритетов оказалась настолько важной для ОСРВ, что реализацию в системе механизмов за-

щиты от этой проблемы вынесли в отдельное функциональное требование к операционным системам реального времени [3]. Инверсия приоритетов возникает, когда два потока, высоко приоритетный (В) и низкоприоритетный (Н) разделяют некий общий ресурс (Р). Предположим, также что в системе присутствует третий поток, приоритет которого находится между приоритетами В и Н. Назовем его средним (С). Если поток В переходит в состояние готовности, когда активен поток Н, и Н заблокировал ресурс Р, то поток В вытеснит поток Н, и Р останется заблокирован. Когда В понадобится ресурс Р, то он сам перейдет в заблокированное состояние. Если в состоянии готовности находится только поток Н, то ничего страшного не произойдет, Н освободит заблокированный ресурс и будет вытеснен потоком В. Но если на момент блокирования потока В в состоянии готовности находится поток С, приоритет которого выше чем у Н, то активным станет именно он, а Н опять будет вытеснен, и получит управление только после того, как С закончит свою работу. Подобная задержка вполне может привести к тому, что критическое время обслуживания потока В будет пропущено. Если В — это поток жесткого реального времени, то подобная ситуация недопустима.

Какие же механизмы защиты от этой проблемы используют разработчики операционных систем реального времени? Наиболее широко распространенный и проверенный механизм — это ***наследование приоритетов***.

Суть этого метода заключается в наследовании низкоприоритетным потоком, захватившим ресурс, приоритета от высокоприоритетного потока, которому этот ресурс нужен. В описанном примере это означает следующее. Если Н блокировал ресурс Р, который нужен В, то при блокировании В его приоритет присваивается потоку Н, и, таким образом, он не может быть вытеснен потоком, с меньшим чем у В приоритетом. После того, как поток Н разблокирует ресурс Р, его приоритет понижается до исходного значения и он вытесняется потоком В.

Механизм наследования приоритетов, к сожалению, не всегда может решить проблемы, связанные с блокированием высокоприоритетного потока на заблокированном ресурсе. В случае, когда несколько средне и низкоприоритетных потоков раз-

деляют некоторые ресурсы с высокоприоритетным потоком, возможна ситуация, когда высокоприоритетному потоку придется слишком долго ждать, пока каждый из младших потоков не освободит свой ресурс, и критический срок обслуживания будет потерян. Однако такие ситуации (разделения ресурсов высокоприоритетного потока) должны отслеживаться разработчиками прикладной системы. В принципе наследование приоритетов является наиболее распространенным механизмом защиты от проблемы инверсии приоритетов.

Другой, несколько менее распространенный метод, называется ***Протокол Предельного Приоритета (Priority Ceiling Protocol).*** Метод заключается в добавлении к стандартным свойствам объектов синхронизации параметра, определяемого максимальным приоритетом потока, которые к объекту обращаются. Если такой параметр установлен, то приоритет любого потока, обращающегося к данному объекту синхронизации, будет увеличен до указанного уровня, и, таким образом, не сможет быть вытеснен никаким потоком, который может нуждаться в заблокированном им ресурсе. После разблокирования ресурса, приоритет потока понижается до начального уровня. Таким образом, получается нечто вроде предварительного наследования приоритетов. Однако этот метод имеет ряд серьезных недостатков. В первую очередь, на разработчика ложится работа по "обучению" объектов синхронизации их уровню приоритетов. Во вторых, возможны задержки в запуске высокоприоритетных потоков на время отработки низкоприоритетных потоков. В целом, максимально эффективно этот механизм может быть использован в случае, когда имеется один поток жесткого реального времени и несколько менее приоритетных потоков, разделяющих с ним ресурсы.

3.1.6. Временные характеристики ОС

Общепринятым условием, отделяющим ОСРВ от операционных систем общего назначения, является следующее: ***Время реакции операционной системы при любых вариантах загрузки должно оставаться постоянным***. На практике это

означает высокую стабильность таких характеристик системы, как латенция прерываний (т.е. время от момента инициации прерывания до первой команды программного обработчика), время переключения контекстов процессов и потоков, и т.д. Также для ОСРВ очень важны времена разрешения конфликтов, таких как приход низкоприоритетного и высокоприоритетного прерываний подряд в указанном порядке с небольшим временным разрывом. Стабильно малое время, за которое управление будет передано обработчику высокоприоритетного прерывания, является хорошей характеристикой ОСРВ. Однако здесь необходимо отметить важный момент: само по себе время реакции системы не играет особой роли, временные характеристики должны рассматриваться в контексте параметров внешнего процесса. Следует помнить, что в системах реального времени ключевыми являются не статистические (средние) оценки, а максимальные значения, поскольку превышение времени реакции даже в одном случае из миллиона в задачах жесткого реального времени Может привести к катастрофическим последствиям.

Еще одна важная особенность операционных систем реального времени, отделяющая их от систем общего назначения, заключается в ***независимости поведения системы и ее времен реакций от количества текущих задач***. В большинстве систем общего назначения такие параметры как время переключения контекста потока прямо зависит от количества потоков в системе, в системах же реального времени этой зависимости быть не должно.

3.1.7. Принципиальные отличия ОСРВ от ОС общего назначения.

В заключение рассмотрения функциональных требований приведем предложения, подчеркивающие принципиальное отличие операционных систем реального времени от операционных систем общего назначения [4]:

1. ОС общего назначения (системы разделения времени), особенно многопользовательские, такие как UNIX, ориентированы на оптимальное распределение ресурсов компьютера между пользователями и задачами. В операционных системах ре-

ального времени подобная задача отходит на второй план — все отступает перед главной задачей — успеть среагировать на события, происходящие на объекте.

2. Другое отличие — применение операционной системы реального времени всегда связано с аппаратурой, с объектом, с событиями, происходящими на объекте. Система реального времени, как аппаратно-программный комплекс, включает в себя датчики, регистрирующие события на объекте, модули ввода-вывода, преобразующие показания датчиков в цифровой вид, пригодный для обработки этих показаний на компьютере, и, наконец, компьютер с программой, реагирующей на события, происходящие на объекте. Операционная система реального времени ориентирована на обработку внешних событий. Именно это приводит к коренным отличиям (по сравнению с ОС общего назначения) в структуре системы, в функциях ядра, в построении системы ввода-вывода. Операционная система реального времени может быть похожа по пользовательскому интерфейсу на ОС общего назначения (к этому, кстати, стремятся почти все производители операционных системах реального времени), однако устроена она иначе.

3. Кроме того, применение операционных системах реального времени всегда конкретно. Если ОС общего назначения обычно воспринимается пользователями (не разработчиками) как уже готовый набор приложений, то операционная система реального времени служит только инструментом для создания конкретного аппаратно-программного комплекса реального времени. И поэтому наиболее широкий класс пользователей операционных системах реального времени — разработчики комплексов реального времени, люди проектирующие системы управления и сбора данных. Проектируя и разрабатывая конкретную систему реального времени, программист всегда знает точно, какие события могут произойти на объекте, знает критические сроки обслуживания каждого из этих событий.

4. Одно из коренных внешних отличий операционных систем реального времени от операционных систем общего назначения — четкое разграничение систем разработки и систем исполнения. Система исполнения операционных системах реального времени — это набор инструментов (ядро, драйверы, ис-

полняемые модули), обеспечивающих функционирование приложения реального времени.

3.2. Архитектуры построения ОСРВ

3.2.1. Монолитные ОС

За свою историю архитектура операционных систем претерпела значительное развитие. Один из первых принципов построения, так называемых монолитных ОС (рис. 3.6), заключался в представлении ОС как набора модулей, взаимодействующих между собой различным образом внутри ядра системы и предоставляющих прикладным программам входные интерфейсы для обращений к аппаратуре.

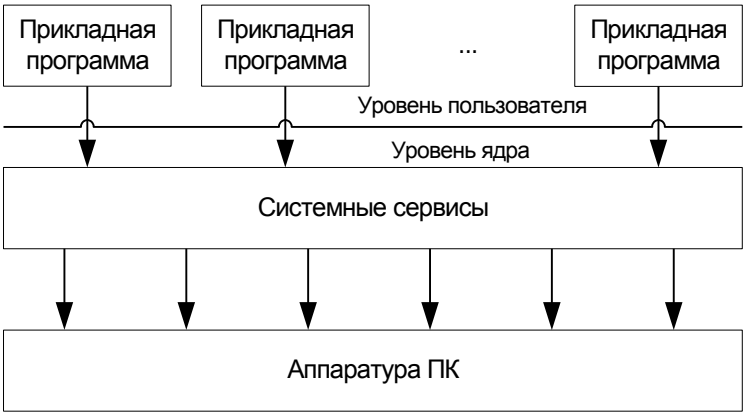

Рис. 3.6. Архитектура монолитной ОС

Системный уровень (уровень ядра) состоит из трех частей:

1. Интерфейс между приложениями и ядром (API — application program interface).
2. Собственно ядро системы.
3. Интерфейс между ядром и оборудованием.

API в таких системах играет двойную роль:

1. Управляет взаимодействием прикладных процессов и систем.

2. Обеспечивает непрерывность выполнения кода системы (то есть отсутствие переключения задач во время исполнения кода программы).

Основным ***преимуществом монолитной*** архитектуры является ее относительная быстрота работы по сравнению с другими архитектурами. Однако это достигается написанием значительной части кода системы на ассемблере.

Главным недостатком такой архитектуры является плохая предсказуемость ее поведения, вызванная сложным взаимодействием модулей системы между собой [5]. Однако большинство современных ОС, как реального времени, так и общего назначения, строятся именно по этому принципу.

К другим недостаткам можно отнести следующие недостатки:

– Системные вызовы, требующие переключения уровней привилегий (от пользовательской задачи к ядру), должны быть реализованы как прерывания или ловушки (специальный тип исключений), что существенно увеличивает время работы.

– Работа ядра не может быть прервана пользовательской задачей (non-preemptable). Из-за этого высокоприоритетная задача не всегда сможет получить управление в момент работы низкоприоритетной задачи.

– Сложность переноса на новые архитектуры процессоров, из-за ассемблерных вставок.

– Негибкость и сложность развития. Изменение части ядра требует его полной перекомпиляции.

3.2.2. Уровневые ОС

В задачах автоматизации широкое распространение в качестве ОСРВ получили уровневые ОС (рис. 3.7). Примером такой ОС является хорошо известная система MS-DOS. В системах этого класса прикладные приложения могли получить доступ к аппаратуре не только посредством ядра системы или ее резидентных сервисов, но и непосредственно. По такому принципу строились ОСРВ в течение многих лет.

По сравнению с монолитными ОС такая архитектура обеспечивает следующее ***преимущество***: значительно большую

степень предсказуемости реакций системы, а также позволяет осуществлять быстрый доступ прикладных приложений к аппаратуре.

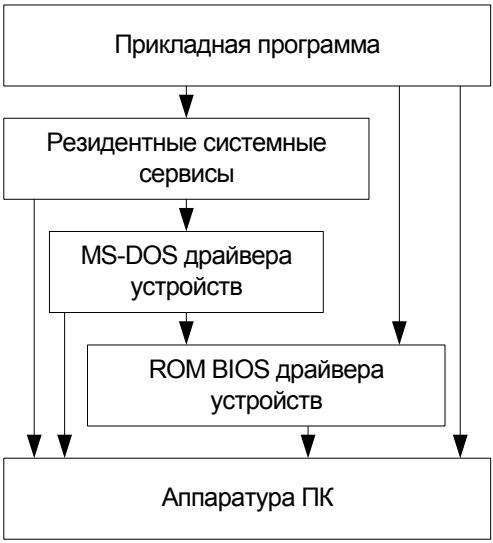

Рис. 3.7. Архитектура уровневой ОС

Недостатком этих систем является отсутствие в них многозадачности. В рамках такой архитектуры проблема обработки асинхронных событий сводилась к буферизации сообщений, а затем последовательному опросу буферов и обработке. При этом соблюдение критических сроков обслуживания обеспечивалось высоким быстродействием вычислительного комплекса по сравнению со скоростью протекания внешних процессов.

3.2.3. Клиент-серверные ОС (модульная архитектура, архитектура на основе микроядра)

Одной из наиболее эффективных архитектур для построения операционных систем реального времени считается архитектура клиент-сервер. Общая схема ОС работающей по этой технологии представлена на рис. 3.8.

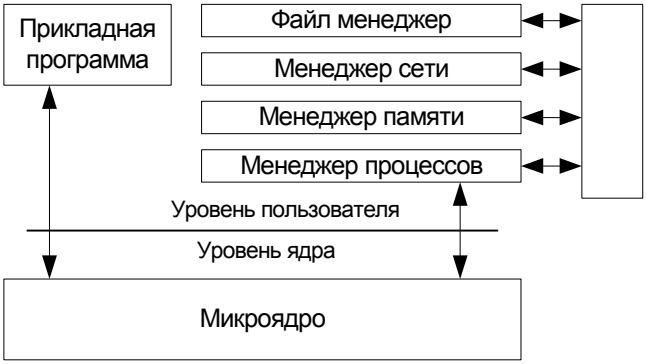

Рис. 3.8. Архитектура клиент-серверной ОС

Основным принципом такой архитектуры является вынесение сервисов ОС в виде серверов на уровень пользователя, а микроядро выполняет функции диспетчера сообщений между клиентскими пользовательскими программами и серверами — системными сервисами. Данная архитектура дает ***преимущества*** с точки зрения требований к ОСРВ и встраиваемым системам:

– Повышается надежность ОС, т.к. каждый сервис является, по сути, самостоятельным приложением, и его легче отладить и отследить ошибки.

– Система лучше масштабируется, поскольку ненужные сервисы могут быть исключены из системы без ущерба для ее работоспособности.

– Повышается отказоустойчивость системы, т.к. «зависший» сервис может быть перезапущен без перезагрузки системы.

Среди известных ОСРВ, реализующих архитектуру микроядра, можно отметить OS9 и QNX.

Недостатки данной архитектуры те же, что и у монолитной архитектуры. Проблемы перешли с уровня API на уровень микроядра. Системный интерфейс по-прежнему не допускает переключения задач во время работы микроядра, только сократилось время пребывания в этом состоянии. Часть API может

также быть реализована на ассемблере. Проблемы переносимости сократились, но остались.

3.2.4. Объектная архитектура на основе объектов-микроядр

В этой архитектуре отсутствует API. Взаимодействие между компонентами системы (микроядрами) и пользовательскими процессами осуществляется посредствам вызова функций. Это возможно если система и приложения написаны на одном языке (C++), что обеспечивает максимальную скорость системных вызовов.

Фактическое равноправие всех компонент системы обеспечивает возможность переключения задач в любое время, т.е. система полностью вытесняема в режиме ядра — preemptable.

Объектно-ориентированный подход полностью обеспечивает модульность, безопасность, легкость модернизации и повторного использования кода.

Роль API играет компилятор и динамический редактор объектных связей (linker). При старте приложения динамический редактор linker загружает нужные ему микроядра. Таким образом в отличии от всех предыдущих типов систем не все компоненты ОС должны быть загружены в оперативную память. Если микроядро уже загружено другим приложением то оно повторно не загружается, а используется код и данные уже имеющегося микроядра. Все это позволяет сократить объем, используемой оперативной памяти. Поскольку разные приложения используют одни микроядра, то они должны работать в одном адресном пространстве. Следовательно, система не может использовать виртуальную память и тем самым работает быстрее. У нее исключаются задержки на трансляцию виртуального адреса в физический.

Основным представителем данной архитектуры является ОСРВ Soft Kernel.

Микроядра по своим характеристикам напоминают структуры, используемые в других операционных системах. Однако есть и различия.

− **Микроядра и модули.** Многие ОС поддерживают динамическую загрузку компонент системы, называемых модулями. Однако, модули не поддерживают объектно-риентрованный подход. Далее, обмен информацией с модулями происходит посредством системных вызовов, что достаточно накладно по времени.
− **Микроядра и драйверы.** Многие ОС поддерживают возможность своего расширения посредством драйверов (специальных модулей, обычно служащих для поддержки оборудования). Однако драйверы часто должны быть статически связаны с ядром (то есть образовывать с ним связанный загрузочный образ еще до загрузки) и должны работать в привилегированном режиме. Далее, как модули они не поддерживают объектно-ориентированный подход и доступны приложению только посредством системных вызовов.
− **Микроядра и DLL.** Многие системы оформляют библиотеки, из которых берутся функции при динамическом связывании, в виде специальных модулей, называемых DLL (Dynamically Linked Libraries — динамически связываемые библиотеки). DLL обеспечивает разделение своего кода и данных для всех работающих приложений, в то время как для микроядер можно управлять доступом для каждого конкретного приложения. DLL не поддерживает объектно-ориентированный подход, код DLL не является позиционно-независимым и не может быть записан в ПЗУ.

3.2.5. Обобщенное построение ОСРВ

Обобщенное построение операционных систем реального времени можно разделить на три слоя (рис. 3.9):
– Ядро — содержит только строгий минимум функций необходимый для работы системы. К минимуму относятся следующие функции:
  - управление, синхронизация и взаимодействие задач;
  - управление памятью;
  - управление устройствами ввода-вывода.

Рис. 3.9. Типичное строение ОСРВ

– Система управления — содержит ядро и ряд дополнительных сервисов, расширяющих его возможности:
  - расширенное управление памятью;
  - расширенное управление устройствами ввода-вывода;
  - расширенное управление задачами;
  - управление файлами;
  - обеспечение взаимодействия системы и внешнего оборудования и т.д.
– Система реального времени — содержит систему управления и набор утилит: средства разработки, средства визуализации и т.д.

3.3. Разделение ОСРВ по способу разработки

Операционные системы реального времени по способу разработки программного обеспечения делят на следующие категории:

Self-Hosted ОСРВ — это системы, в которых пользователи могут разрабатывать приложения, работая в самой ОСРВ. Обычно это предполагает, что ОСРВ поддерживает файловую систему, средства ввода-вывода, пользовательский интерфейс, имеет компиляторы, отладчик, средства анализа программ, текстовые редакторы, работающие под управлением ОСРВ.

Достоинством таких систем является более простой и наглядный механизм создания и запуска приложений, которые работают на той же машине, что и пользователь. Недостатком является то, что промышленному компьютеру во время его реальной эксплуатации часто вообще не требуется пользовательский интерфейс и возможность запуска тяжеловесных программ вроде компилятора. Следовательно, большинство возможностей ОСРВ не используются и только зря занимают память и другие ресурсы компьютера.

Обычно self-hosted ОСРВ применяются на «обычных» компьютерах промышленного исполнения.

Host/Target ОСРВ — это системы, в которых операционная система и компьютер, на котором разрабатываются приложения (host), и операционная система и компьютер, на котором запускаются приложения (target), различны. Связь между компьютерами осуществляется с помощью последовательно соединения (COM-порта), Ethernet, общей шины VME или compact PCI. В качестве host системы обычно выступает компьютер под управлением Unix или Windows NT, в качестве target системы выступает промышленный или встраиваемый компьютер под управлением ОСРВ.

Достоинством таких систем является использование всех ресурсов «обычной» системы для создания приложений и уменьшение размеров ОСРВ за счет включения только нужных приложению компонент. Недостатком является относительная сложность программных компонент: кросс-компилятора, удаленного загрузчика и отладчика и т.д.

В настоящее время, рост мощности промышленных компьютеров позволяет использовать self-hosted ОСРВ на большем числе вычислительных систем. Хотя, с другой стороны, увеличивается распространение встраиваемых систем (в разнообразном промышленном и бытовом оборудовании), расширяет сферу применения host/target систем, поскольку при больших объемах выпуска цена системы является определяющим фактором.

Большинство современных ведущих операционных систем реального времени поддерживают целый спектр аппаратных архитектур, на которых работают системы исполнения (Intel, Motorola, RISC,MIPS, PowerPC и другие). Это объясняется

тем, что набор аппаратных средств — часть комплекса реального времени и аппаратура должна быть также адекватна решаемой задаче, поэтому ведущие операционные системы реального времени перекрывают целый ряд наиболее популярных архитектур, чтобы удовлетворить самым разным требованиям по части аппаратуры.

Заметим, что функционально средства разработки операционных систем реального времени отличаются от привычных систем разработки, так как часто они содержат средства удаленной отладки, средства профилирования (измерение времен выполнения отдельных участков кода), средства эмуляции целевого процессора, специальные средства отладки взаимодействующих задач, а иногда и средства моделирования.

Вопросы для самопроверки

1. Дайте определение понятию механизма диспетчеризации.
2. Расскажите о проблеме разделения ресурсов.
3. В чем отличие понятий процессов от потоков?
4. Какое свойство потока отвечает за предоставление ему процессорного времени?
5. В каких состояниях может находиться поток?
6. Какие методы диспетчеризации Вы знаете?
7. В чем отличие вытесняющей многозадачности от адаптивной и приоритетной?
8. Сколько уровней приоритетов должна иметь операционная система реального времени?
9. Какая серьезная проблема возникает при блокировании ресурсов?
10. Какие механизмы существуют для решения проблемы инверсии приоритетов?
11. Какие требования по временным характеристикам накладываются на системы реального времени?
12. Сформулируйте принципиальные отличия ОСРВ от ОС общего назначения.
13. Перечислите архитектуры построения ОСРВ.
14. Как можно описать обобщенную структуру ОСРВ?

15. В чем отличие категории OCPB Self-Hosted от категории Host/Target?

4. Стандарты на ОСРВ

4.1. Важность стандартов на ОСРВ

Стандарты на операционные системы реального времени стали появляться лишь после того, как уже был создан ряд ОСРВ. Основной целью введения стандартов является облегчение переноса программного обеспечения из одной системы в другую.

Несмотря на стандарты, основной целью разработчиков ОСРВ является обеспечение максимальной скорости ее работы и компактности. Поэтому, с одной стороны, среди ОСРВ преобладают системы с уникальным устройством, а с другой стороны, многие стандарты носят общий характер. При этом даже системы, декларирующие свою совместимость с некоторым стандартом, обычно содержат ряд расширений, выходящих за его рамки. Тем не менее, важность стандартов состоит в том, что они фактически выступают в качестве аксиоматической базы, задающей определения рассматриваемых объектов и понятий.

4.2. Стандарт SCEPTRE

Стандарт SCEPTRE (Standardisation du Cœur des Exécutifs des Produits Temps Réel Européens) — европейский стандарт на основы систем реального времени (sceptre по-французски означает «скипетр») разрабатывался в 1980-90 годы. За время его создания появились новые концепции в ОСРВ, не все из которых успели найти отражение в стандарте. В стандарте объединены усилия инженеров и исследователей в разработке групп спецификаций для промышленных приложений, даны определения и описания набора методов и подходов, используемых в ОСРВ.

Стандарт SCEPTRE определяет семь основных целей, которые должны преследовать ОСРВ:

1. Адекватность поставленной задаче.

2. Безопасность (система должна быть максимально устойчивой к аппаратным и программным сбоям).
3. Минимальная стоимость.
4. Максимальная производительность.
5. Переносимость (возможность реализовывать систему на другом типе процессора, адекватном поставленной задаче).
6. Адаптивность (способность системы приспосабливаться к новому управляемому ею оборудованию и задачам).
7. Модульность (система должна состоять из достаточно независимых компонент, из которых можно построить систему, достаточную для решения поставленной задачи на имеющемся оборудовании).

Весь сервис, предоставляемый операционной системой, разделен в стандарте на следующие группы:

– коммуникации (межпроцессорное взаимодействие);
– синхронизация процессов;
– контроль и планирование задач;
– управление памятью;
– управление прерываниями и оборудованием ввода/вывода;
– высокоуровневый интерфейс ввода/вывода и управление периферийными устройствами;
– управление файлами;
– управление транзакциями (сообщениями и передачами данных);
– обработка ошибок и исключений;
– управление временем.

4.3. Стандарт POSIX

Наиболее ранним и распространенным стандартом ОСРВ является стандарт ***POSIX (IEEE Portable Operating System Interface for Computer Environments, IEEE 1003.1).*** Первоначальный вариант стандарта POSIX появился в 1990 г. и был предназначен для UNIX-систем, первые версии которых появились в 70-х годах прошлого века. Спецификации POSIX определяют стандартный механизм взаимодействия прикладной программы и операционной системы и в настоящее время включа-

ют набор более чем из 30 стандартов. Для ОСРВ наиболее важны семь из них (1003.1a, 1003.1b, 1003.1c, 1003.1d, 1003.1j, 1003.21, 1003.2h), но широкую поддержку в коммерческих ОС получили только три первых.

Несмотря на явно устаревшие положения стандарта POSIX и большую потребность в обновлении стандартизации для ОСРВ, заметного продвижения в этом направлении не наблюдается.

Стандарт POSIX был создан как стандартный интерфейс сервисов операционных систем. Этот стандарт дает возможность создавать переносимые приложения. Впоследствии этот стандарт был расширен особенностями режима реального времени.

Спецификации POSIX задают стандартный механизм взаимодействия приложения и ОС.

Несмотря на то, что стандарт POSIX вырос из Unix, он затрагивает основополагающие абстракции операционных систем, а расширения реального времени применимы ко всем ОСРВ.

К настоящему времени стандарт POSIX рассматривается как семейство родственных стандартов: IEEE Std 1003.n (где n – это номер).

Стандарт 1003.1a (OS Definition) содержит базовые интерфейсы ОС — поддержку единственного процесса, поддержку многих процессов, управление заданиями, сигналами, группами пользователей, файловой системой, файловыми атрибутами, управление файловыми устройствами, блокировками файлов, устройствами ввода/вывода, устройствами специального назначения, системными базами данных, каналами, очередями FIFO, а также поддержку языка С.

Стандарт 1003.1b (Realtime Extensions) содержит расширения реального времени:

А. Диспетчеризация процессов реального времени.

В базовом стандарте POSIX определяется модель параллельной работы процессов, но никакого механизма диспетчеризации и никакой концепции приоритетов не специфицируется. Для приложений реального времени необходимо специфицировать некоторый механизм диспетчеризации, удовлетворяющий специфике реального времени. В стандарте 1003.1b специфици-

руется три механизма диспетчеризации. У каждого процесса есть атрибут диспетчеризации, который должен устанавливаться в соответствии с одним из трех механизмов [6]:

– SCHED_FIFO: это механизм приоритетной диспетчеризации с фиксированными приоритетами, при которой процессы с одинаковыми приоритетами обрабатываются по принципу "первым пришел — первым вышел" (FIFO). Этот механизм должен обслуживать минимум 32 уровня приоритетов.

– SCHED_RR: этот механизм подобен механизму SCHED__FIFO, но в нем для диспетчеризации равноприоритетных процессов используется метод квантования времени (круговая диспетчеризация). И этот метод обладает 32 уровнями приоритетов.

– SCHED_OTHER: этот механизм диспетчеризации определяется конкретной реализацией.

Используя эти методы диспетчеризации, а также функции установки приоритета каждого процесса и функции включения (разрешения) нужного метода, в операционных системах в стандарте POSIX можно выполнять диспетчеризацию приложений реального времени.

B. Блокирование виртуальной памяти.

Хотя в базовом стандарте POSIX использование виртуальной памяти не требуется, в UNIX-системах этот механизм широко распространен. Он очень эффективен при работе программ, не относящихся к программам реального времени, но приводит к непредсказуемости времени реакции системы. Для того чтобы ограничить время доступа к памяти, в стандарте 1003.1b определяются функции блокировки (фиксации) в памяти всего адресного пространства процесса или отдельных его областей. Эти функции следует использовать для критичных ко времени процессов, а также для процессов, с которыми синхронизируются критичные ко времени процессы. В этом случае время их реакции может быть предсказуемым.

C. Синхронизация процессов.

В стандарте 1003.1b определяются функции управления синхронизацией процессов с помощью семафоров-счетчиков. Эти семафоры идентифицируются по имени, находящемся в некотором пространстве имен, определяемом при реализации стандарта. Это пространство имен может совпадать, но не обязательно, с пространством имен файлов. Семафор-счетчик — это общий механизм синхронизации, который позволяет реализовать взаимно исключающий доступ к разделяемым ресурсам, передачу сигналов, ожидание процессов и другие механизмы синхронизации. Одним из наиболее распространенных применений семафоров является совместное использование данных процессами. Согласно стандарту, это можно реализовать с помощью объектов разделяемой памяти, используемых вместе с семафорами.

К сожалению, семафоры-счетчики, определение которых дается в стандарте, допускают инверсию приоритетов. Она возникает тогда, когда высокоприоритетный процесс вынужден ждать завершения некоторого действия, выполняемого процессом с более низким приоритетом. Это означает, что задержка высокоприоритетных задач вызывается не длительностью выполнения критических секций, а зависит от общего времени выполнения низкоприоритетных задач. Эта ситуация может возникнуть тогда, когда высокоприоритетная задача ожидает освобождения низкоприоритетной задачей семафора, который управляет доступом к разделяемым ресурсам, а низкоприоритетная задача вытеснена задачей с промежуточным значением приоритета. Для задач жесткого реального времени такие длительные задержки обычно неприемлемы. Поэтому в стандарте 1003.1c (Threads) описывается другой механизм синхронизации — мьютекс (mutex), который предотвращает незаконное отрицание приоритета и также может использоваться в процессах.

D. Разделяемая память.

В соответствии с базовым стандартом POSIX процессы имеют независимые адресные пространства, но во многих приложениях реального времени (и других) требуется совместное использование, с очень малыми издержками, большого количе-

ства данных. Это возможно в случае, если процессы могут разделять части физической памяти. В стандарте 1003.1b определяются объекты разделяемой памяти, представляющие собой участки физической памяти, которые могут отображаться на адресное пространство процесса. Когда два или несколько процессов отображают один и тот же объект, они разделяют связанный с ним участок памяти. Как и в случае семафоров, объекты разделяемой памяти идентифицируются по имени, принадлежащему некоторому пространству имен, которое определяется при реализации стандарта. Если к объектам данных, находящихся в разделяемой памяти, нужен взаимно исключающий доступ, для управления этим доступом можно использовать семафоры. В адресное пространство процесса могут отображаться и файлы — для совместного использования с другими процессами.

E. Сигналы реального времени.

Механизм сигналов, определяемый в базовом стандарте POSIX, позволяет извещать о событиях, возникающих в системе, но требования приложений реального времени полностью удовлетворить не может. Сигналы не могут образовывать очереди, поэтому некоторые события могут теряться. Сигналы не могут иметь приоритетов, из-за чего возможно увеличение времени отклика на события, требующие неотложной реакции. Кроме того, события одного вида вызывают сигналы с одинаковым номером, которые не различаются. Так как многие системы реального времени должны обеспечивать быстрый обмен событиями, интерфейс сигналов в стандарте 1003.1b расширен, чтобы получить следующие возможности:

Сигналы реального времени устанавливаются в очередь, поэтому события не теряются.

Необработанные сигналы реального времени извлекаются из очереди по приоритетам, где в качестве приоритета служит номер сигнала. Это предоставляет возможность быстрого отклика на события, требующие неотложной реакции.

Сигналы реального времени содержат дополнительное поле данных, которое может использоваться прикладной системой для обмена между генератором сигнала и его обработчиком.

Например, это поле данных может использоваться для идентификации источника сигнала.

Расширен диапазон доступных прикладной системе сигналов.

F. Взаимодействие процессов.

Для взаимодействия процессов определяется простой механизм очередей сообщений. Очереди сообщений идентифицируются по имени, принадлежащему некоторому пространству имен, определяемому при реализации стандарта. Сообщения имеют связанное с ними поле приоритета и извлекаются из очереди в соответствии с приоритетом. Это способствует сокращению незаконного отрицания приоритета, возникающего в системе. Передача и получение сообщений может блокироваться и разблокироваться; передача и получение не синхронизируются, то есть, отправитель не ждет, когда получатель действительно извлечет сообщение из очереди. Максимальный размер сообщений и очередей определяется пользователем, а необходимые для поддержания очереди ресурсы могут выделяться заранее во время разработки приложения, что повышает предсказуемость работы с очередями сообщений.

G. Часы и таймеры.

Определяются часы реального времени, которые измеряют время с точностью настенных часов. Эти часы должны обеспечивать разрешение минимум 20 мс. Так как время представляется с наносекундным разрешением, разработчики могут воспользоваться высокоточными аппаратными часами. Для отсчета временных интервалов на основе часов реального времени или других часов, определенных при реализации стандарта, могут создаваться таймеры. По истечении заданного интервала времени эти таймеры генерируют сигнал, направленный процессу, создавшему данный таймер. Существует несколько опций, таких, как периодическая сигнализация, единичный сигнал и т. д., которые позволяют легко реализовать, например, генерацию периодических событий. Для приостановки вызывающего процесса на некоторый заданный период времени определяется функция относительного сна (nanosleep).

H. Асинхронный ввод/вывод

В стандарте 1003.1b определяются функции, которые обеспечивают возможность совмещать прикладную обработку и операции ввода/вывода, инициированные данным приложением. Асинхронные операции ввода/вывода подобны обычным операциям, за исключением того, что после того как процесс инициировал асинхронную операцию ввода/вывода, он продолжает выполняться параллельно этой операции. Когда операция завершается, данному приложению может быть послан сигнал.

I. Другие функции.

В стандарте 1003.1b определяются другие функции, такие, как синхронизированный ввод/вывод, файлы реального времени и т. п.

Стандарт 1003.1c (Threads) касается функций поддержки многопоточной обработки внутри процесса:

A. Управление потоками.

Эти функции позволяют управлять созданием и завершением выполнения потоков, а также связанными с ними операциями. Определяются функции создания потока, ожидания завершения потока, нормального завершения потока, открепления потока (то есть указания разработчику, что связанная с потоком память может после завершения потока перераспределяться) или создания конкретной потока только в том случае, если он еще не был создана. Другие функции позволяют управлять идентификаторами потоков. Определяются также функции для управления атрибутами при создании потоков, такими, как размер стека, возможность перераспределения памяти, занимаемой потоком после ее создания, и т.п.

B. Диспетчеризация потоков.

Для потоков определяются те же методы диспетчеризации, что и для процессов в стандарте 1003.1b. Так как в системе могут одновременно существовать два планировщика (диспетчера): планировщик процессов и планировщик потоков. Конку-

рентное пространство некоторого потока — это набор потоков, с которыми он конкурирует за центральный процессор. При реализации может возникнуть три основных вида различных конкурентных пространств:

Глобальная Диспетчеризация. Все потоки имеют глобальное конкурентное пространство, и, следовательно, диспетчеризация каждого потока производится с учетом всех остальных потоков, имеющихся в системе, независимо от того, какому процессу они принадлежат. Планировщик работает только на уровне потоков, а параметры диспетчеризации процессов игнорируются.

Локальная Диспетчеризация. Потоки конкурируют только с другими потоками того же самого процесса. Диспетчеризация производится на двух уровнях. Сначала подвергаются диспетчеризации все процессы по отношению друг к другу. Затем потоки выбранного процесса конкурируют друг с другом за центральный процессор.

Смешанная Диспетчеризация. Некоторые потоки имеют глобальное конкурентное пространство, а другие — локальное. Диспетчеризация производится на двух уровнях: на первом уровне диспетчеризации подвергаются процессы и глобальные потоки; на втором уровне происходит диспетчеризация локальных потоков выбранного процесса.

Наилучшие результаты для приложений реального времени дает глобальная и смешанная диспетчеризация, так как в этих случаях можно производить диспетчеризацию на одном и том же уровне всех, самых разных, конкурирующих объектов, обладающих жесткими временными требованиями. В системах со смешанной диспетчеризацией диспетчеризация отдельных потоков может осуществляться и локально. Локальная диспетчеризация быстрее и эффективнее глобальной. Однако эту возможность следует использовать только для групп потоков, глобальный приоритет которых меньше или больше приоритетов остальных групп потоков, имеющихся в системе (то есть, когда ни один из других потоков в системе не имеет уровня приоритета, попадающего между уровнями приоритетов каких-либо потоков этой группы). Причина заключается в том, что для диспетчеризации группы потоков с локальным конкурентным про-

странством будет использоваться приоритет процесса, а не приоритеты потоков. Это относится и к системам с локальной диспетчеризацией.

С. Синхронизация потоков

Для потоков определяется два примитива синхронизации: мьютексы (mutex, mutually exlusive) и условные переменные (condvars, condition variables). Мьютексы используются для синхронизации взаимно исключающего доступа потоков к разделяемым ресурсам, а условные переменные используются для сигнализации и ожидания событий среди потоков. Ожидание условной переменной, с которой должен быть связан сигнал, можно задать с помощью таймаута. Опционально эти примитивы можно использовать для нитей, относящихся к разным процессам.

Мьютексы определяются с помощью трех альтернативных протоколов синхронизации:

– NO_PRIO_INHERIT. Приоритет потока не зависит от ее владения мьютексами (мьютексом владеет данная нить, которая его заблокировала);

– PRIOJNHERIT. Поток, владеющий некоторым мьютексом, наследует приоритеты потока, ожидающих захвата этого мьютекса. Это называется протоколом наследования приоритетов;

– PRIO_PROTECT. Когда поток блокирует захваченный им мьютекс, он наследует приоритет, соответствующий данному мьютексу. Этот приоритет определяется прикладной программой как атрибут мьютекса. В случае использования соответствующего значения, равного наименьшему целому числу, превышающему значения приоритетов, получаем протокол приоритетной защиты, называемый также эмуляцией протокола с перекрытием приоритета.

Используя один из двух последних протоколов, можно избежать инверсии приоритетов, достигая, таким образом, высокой эффективности работы системы с жесткими требованиями к характеристикам реального времени. Протокол приоритетной защиты вместе с подходящими определениями значения пере-

крытия приоритета можно использовать и для избежания особого вида инверсии приоритетов, возникающего в многопроцессорной среде и называемого удаленным блокированием.

D. Другие функции.

Для управления специфическими данными, связанными с потоками, для завершения потоков, для посылки сигналов потокам и управления реентерабельными функциями, в стандарте 1003.1c определяются и другие функции.

Стандарт 1003.1d включает поддержку дополнительных расширений реального времени — семантика порождения новых процессов (spawn), спорадическое серверное планирование, мониторинг процессов и потоков времени выполнения, таймауты функций блокировки, управление устройствами и прерываниями.

Стандарт 1003.21 касается распределенных систем реального времени и включает функции поддержки распределенного взаимодействия, организации буферизации данных, посылки управляющих блоков, синхронных и асинхронных операций, ограниченной блокировки, приоритетов сообщений, меток сообщений, и реализаций протоколов.

Стандарт 1003.2h касается сервисов, отвечающих за работоспособность системы.

4.4. DO-178B

Стандарт DO-178 «Software Consideration in Airborne Systems and Equipment Certification» разработан и поддерживается ассоциацией RTCA (Radio Technical Commission for Aeronautics, www.rtca.org). Первая его версия была принята в 1982 г., вторая (DO-178A) — в 1985-м, текущая DO-178B — в 1992 г., а принятие новой версии DO-178C затягивается уже несколько лет, в прессе сообщается, что он должен быть принят в течение 2010 года. Стандартом предусмотрено пять уровней серьезности отказа, и для каждого из них определен набор требований к программному обеспечению, которые должны гарантировать рабо-

тоспособность всей системы в целом при возникновении отказов данного уровня серьезности:

− уровень A — программное обеспечение должно обеспечивать защиту от сбоев, приводящих к катастрофическим (catastrophic) последствиям, и удовлетворять 66 требованиям;
− уровень B — программное обеспечение должно обеспечивать защиту от сбоев, приводящих к опасным (hazardous) последствиям, и удовлетворять 65 требованиям;
− уровень C — программное обеспечение должно обеспечивать защиту от сбоев, приводящих к серьезным (major) последствиям, и удовлетворять 57 требованиям;
− уровень D — программное обеспечение должно обеспечивать защиту от сбоев, приводящих к незначительным (minor) последствиям, и удовлетворять 28 требованиям;
− уровень E — программное обеспечение должно обеспечивать защиту от сбоев, не приводящих ни к каким последствиям.

До тех пор пока все жесткие требования этого стандарта не будут выполнены, вычислительные системы, влияющие на безопасность, никогда не поднимутся в воздух.

4.5. ARINC-653

Стандарт ARINC-653 (Avionics Application Software Standard Interface) разработан компанией ARINC в 1997 г. Этот стандарт определяет универсальный программный интерфейс APEX (Application/Executive) между ОС авиационного компьютера и прикладным программным обеспечением. Требования к интерфейсу между прикладным программным обеспечением и сервисами операционной системы определяются таким образом, чтобы разрешить прикладному программному обеспечению контролировать диспетчеризацию, связь и состояние внутренних обрабатываемых элементов. В 2003 г. принята новая редакция этого стандарта. ARINC-653 в качестве одного из основных требований для ОСРВ в авиации вводит архитектуру изолированных (partitioning) виртуальных машин [6].

4.6. OSEK

Стандарт OSEK/VDX является комбинацией стандартов, которые изначально разрабатывались в двух отдельных консорциумах, впоследствии слившихся. OSEK берет свое название от немецкого акронима консорциума, в состав которого входили ведущие немецкие производители автомобилей – BMW, Bosch, Daimler Benz (теперь Daimler Chrysler), Opel, Siemens и Volkswagen, а также университет в Карлсруэ (Германия). Проект VDX (Vehicle Distributed eXecutive) развивался совместными усилиями французских компаний PSA и Renault. Команды OSEK и VDX слились в 1994г [6].

Первоначально проект OSEK/VDX предназначался для разработки стандарта открытой архитектуры ОС и стандарта API для систем, применяющихся в автомобильной промышленности. Однако разработанный стандарт получился более абстрактным и не ограничивается использованием только в автомобильной индустрии [6].

Стандарт OSEK/VDX состоит из трех частей [6]:

- стандарт для операционной системы (OS);
- коммуникационный стандарт (COM);
- стандарт для сетевого менеджера (NM).

В дополнение к этим стандартам определяется некий реализационный язык (OIL). Первым компонентом стандарта OSEK является стандарт для ОС, поэтому часто стандарт OSEK ошибочно воспринимается как стандарт ОСРВ. Хотя ОС и есть большая порция данного стандарта, мощность его состоит в интеграции всех его компонент.

4.7. Стандарты безопасности

В связи со стандартами для ОСРВ стоит отметить широко известный стандарт критериев оценки пригодности компьютерных систем (TCSEC — Trusted Computer System Evaluation Criteria). Этот стандарт разработан Министерством обороны США и известен также под названием «Оранжевая книга».

В ряде других стран были разработаны аналогичные критерии, на основе которых был создан международный стандарт «Общие критерии оценки безопасности информационных технологий» (Common Criteria for IT Security Evaluation, ISO/IEC 15408).

В «Оранжевой книге» перечислены семь уровней защиты [6]:

– A1 — верифицированная разработка. Этот уровень требует, чтобы защиту секретной и другой критичной информации средствами управления безопасностью гарантировали методы формальной верификации.
– B3 — домены безопасности. Этот уровень предназначен для защиты систем от опытных программистов.
– B2 — структурированная защита. В систему с этим уровнем защиты нельзя допустить проникновение хакеров.
– B1 — мандатный контроль доступа. Защиту этого уровня, возможно, удастся преодолеть опытному хакеру, но никак не рядовым пользователям.
– C2 — дискреционный контроль доступа. Уровень C2 обеспечивает защиту процедур входа, позволяет производить контроль за событиями, имеющими отношение к безопасности, а также изолировать ресурсы.
– C1 — избирательная защита. Этот уровень дает пользователям возможность защитить личные данные или информацию о проекте, установив средства управления доступом.
– D — минимальная защита. Этот нижний уровень защиты оставлен для систем, которые проходили тестирование, но не смогли удовлетворить требованиям более высокого класса.

Что касается общих критериев, то в них введены похожие требования обеспечения безопасности в виде оценочных уровней (EAL — Evaluation Assurance Levels). Их также семь [6]:

– EAL7 — самый высокий уровень предполагает формальную верификацию модели объекта оценки. Он применим к системам очень высокого риска.
– EAL6 — определяется, как полуформально верифицированный и протестированный. На уровне EAL6 реализация должна быть представлена в структурированном виде, анализ соответ-

ствия распространяется на проект нижнего уровня, проводится строгий анализ покрытия, анализ и тестирование небезопасных состояний.

– EAL5 — определяется, как полуформально спроектированный и протестированный. Он предусматривает создание полуформальной функциональной спецификации и проекта высокого уровня с демонстрацией соответствия между ними, формальной модели политики безопасности, стандартизованной модели жизненного цикла, а также проведение анализа скрытых каналов.

– EAL4 — определяется, как методически спроектированный, протестированный и пересмотренный. Он предполагает наличие автоматизации управления конфигурацией, полной спецификации интерфейсов, описательного проекта нижнего уровня, подмножества реализаций функций безопасности, неформальной модели политики безопасности, модели жизненного цикла, анализ валидации, независимый анализ уязвимостей. По всей вероятности, это самый высокий уровень, которого можно достичь на данном этапе развития технологии программирования с приемлемыми затратами.

– EAL3 — определяется, как методически протестированный и проверенный. На уровне EAL3 осуществляется более полное, чем на уровне EAL2, тестирование покрытия функций безопасности, а также контроль среды разработки и управление конфигурацией объекта оценки.

– EAL2 — определяется, как структурно протестированный. Он предусматривает создание описательного проекта верхнего уровня объекта оценки, описание процедур инсталляции и поставки, руководств администратора и пользователя, функциональное и независимое тестирование, оценку прочности функций безопасности, анализ уязвимостей разработчиками.

– EAL1 — определяется, как функционально протестированный. Он обеспечивает анализ функций безопасности с использованием функциональной спецификации и спецификации интерфейсов, руководящей документации, а также независимое тестирование. На этом уровне угрозы не рассматриваются как серьезные.

В соответствии с требованиями Общих критериев, продукты определенного класса (например, операционные системы) оцениваются на соответствие ряду функциональных критериев и критериев доверия — профилей защиты. Существуют различные определения профилей защиты в отношении операционных систем, брандмауэров, смарт-карт и прочих продуктов, которые должны соответствовать определенным требованиям в области безопасности. Например, профиль защиты систем с разграничением доступа (***Controlled Access Protection Profile***) действует в отношении операционных систем и призван заменить старый уровень защиты C2, определявшийся в соответствии с американским стандартом TCSEC. В соответствии с оценочными уровнями доверия сертификация на соответствие более высокому уровню означает более высокую степень уверенности в том, что система защиты продукта работает правильно и эффективно, и, согласно условиям Общих критериев, уровни 5-7 рассчитаны на тестирование продуктов, созданных с применением специализированных технологий безопасности [6].

Следует отметить, что большинство усилий по оценке продуктов безопасности сосредоточены на уровне 4 стандарта Общих критериев и ниже, что говорит об ограниченном применении формальных методов в этой области [6].

С точки зрения программиста Общие критерии можно рассматривать как набор библиотек, с помощью которых пишутся задания по безопасности, типовые профили защиты и т.п [6].

Вопросы для самопроверки

1. Чем обусловлена важность стандартов на ОСРВ?
2. Какие основные цели определяет стандарт SCEPTRE?
3. Опишите стандарт POSIX 1003.1b.
4. Опишите стандарт POSIX 1003.1c.
5. Опишите пять уровней серьезности отказа стандарта DO-178B.
6. Что определяет стандарт ARINC-653?
7. Из каких основных частей состоит стандарт OSEK/VDX?
8. Для чего предназначен стандарт TCSEC?

9. Сколько уровней защиты перечислены в «Оранжевой книге»?
10. Какие уровни общих критериев предусмотрены стандартом TCSEC?

5. Обзор ОСРВ

5.1. Классификация ОСРВ в зависимости от происхождения

В зависимости от происхождения ОСРВ разделяют на следующие группы:

Обычные ОС, используемые в качестве ОСРВ. Часто к обычным ОС добавляют дополнительные модули, реализующие поддержку специфического оборудования, а также планирования задач и обработку прерываний в соответствие с требованиями к ОСРВ и сглаживающие невозможность прервать ядро системы. Все системы относятся к разряду self-hosted.

Собственно ОСРВ. Бывают как self-hosted, так и host/target, некоторые ОСРВ поддерживают обе модели.

Специализированные ОСРВ. Это ОСРВ разрабатываемые для конкретного микроконтроллера его производителем. Часто не являются полноценными ОС, а представляют единый модуль с приложением и обеспечивают только необходимый минимум функций. Все такие системы относятся к категории host/target.

5.2. Системы на основе обычных ОС

5.2.1. Linux

Системы на основе Linux (свободно распространяемой версии Unix). Она получила значительное распространение на настольных компьютерах благодаря своей бесплатности и качества.

В настоящий момент времени система Linux может работать помимо процессоров Intel 80x86 на процессорах Alpha, SPARC, PowerPC, ARM, Motorola 68xxx, MIPS. Открытость ее исходных текстов позволяет реализовывать на ее основе специализированные системы и обеспечивать поддержку нового оборудования.

Приспособление системы Linux к требованиям реального времени происходит по следующим направлениям:

Поддержка стандартов POSIX, касающихся реального времени. Стандарт POSIX 1003.1c (работа с задачами) уже поддержан, Стандарт POSIX 1003.1b (работа расширения реального времени) поддержан лишь частично: реализованы механизмы управления памятью и механизмы планирования задач.

Поддержка специального оборудования, важнейшим из которых является шина VME. Существует поддержка моста VME-PCI. Также для системы реального времени важным является повышение разрешения таймера системы.

Реализация механизма preemption (приоритетное прерывание обслуживания) для ядра системы. Этот механизм является необходимым для того, чтобы систему можно было назвать системой реального времени, но он также является очень сложным для реализации. Linux надолго запрещает прерывания при входе в ядро системы и не является preemptive.

Существует несколько проектов реализации ***preemption*** для ядра Linux. По способу решения этой задачи их можно разделить на две группы:

1. Механизм preemption реализуется путем переписывания ядра системы. На этом пути можно достичь самых качественных результатов, но на данный момент времени значительных успехов в этом плане нет по следующим причинам:

– слишком большой объем работы, связанный с большим объемом ядра;

– слишком высокая скорость изменения ядра, причем изменения вносятся, не учитывая интересы реального времени.

2. Механизм preemption реализуется путем написания микроядра, отвечающего за диспетчеризацию прерываний и задач. Ядро Linux работает как задача с низким приоритетом. Само ядро лишь незначительно изменено, для предотвращения блокирования им аппаратных прерываний. Задачи в такой системе разделены на две группы:

– процессы, работающие под управлением только микроядра (не использующие функции ядра Linux);

– процессы, работающие под управлением Linux (обычные приложения), а также задачи, работающие под управлением

микроядра, но использующие функции Linux. Это процессы неудовлетворяющие требованиям реального времени, поскольку могут быть блокированы ядром Linux.

Недостатком такого подхода является необходимость реализации микроядра, обеспечивающего функционирование процессов реального времени. Например, если процесс реального времени хочет работать с коммуникационным портом, то драйвер этого порта надо перенести из ядра Linux в микроядро. Наиболее законченной реализацией этого подхода является проект ***RT-Linux***. На основе Linux существуют также расширения ***KURT*** и ***UTIME***, позволяющие получить устойчивую среду реального времени.

RTLinux представляет собой систему «жесткого» реального времени, а KURT (KU Real Time Linux) относится к системам «мягкого» реального времени. Linux-расширение UTIME, входящее в состав KURT, позволяет добиться увеличения частоты системных часов, что приводит к более быстрому переключению контекста задач.

Система RT-Linux является свободно распространяемой, разрабатываемой энтузиастами в ряде университетов мира. Создана в New Mexico Institute of Mining and Technology (USA).

Представляет собой простейшее микроядро, отвечающее за создание и планирование задач, обеспечение их взаимодействия и диспетчеризацию прерываний. Реализован простейший приоритетный механизм планирования и единственный механизм взаимодействия — очередь сообщений FIFO. Ядро Linux работает как самая низкоприоритетная задача. В само ядро внесены исправления — макроопределения, отвечающие за запрещения/разрешения прерываний, заменены на соответствующие функции микроядра. Задачи Linux не могут прервать ядро Linux, задачи же микроядра могут.

Такая структура накладывает некоторые ограничения на задачи реального времени. Они не могут легко использовать различные драйвера Linux, не имеют доступ к сети и т.д. Зато они могут обмениваться данными со стандартами задачами Linux.

Простые очереди FIFO реализованы для обмена данными между процессами реального времени и процессами Linux.

Типичное приложение состоит из двух частей — задачи реального времени, непосредственно работающей с аппаратурой и обыкновенно задачи Linux, которая выполняет остальные операции, такие как сохранение данных на диск, пересылка их по сети, работа с пользователем (GUI) и т.д.

Самый короткий период для периодически вызываемых задач реального времени в RT-Linux на Pentium 120 — менее 150 мкс. Задачи, вызываемые по прерыванию могут иметь намного меньший период.

Ядро реального времени не защищает от перегрузок. Если одна из задач реального времени полностью утилизирует процессор, ядро Linux, имея самый низкий приоритет, не получит управления и система повиснет.

Задачи реального времени запускаются в адресном пространстве ядра и с привилегиями ядра и могут быть реализованы, например, при помощи модулей Linux.

Минимальный размер системы для записи в ПЗУ (без X-Window) — 2.7 Мб.

5.2.2. Windows NT

Системы на основе Windows NT. По поводу использования Windows NT в качестве ОСРВ ведется большое количество дискуссий. Существуют аргументы и «За» и «Против».

Аргументы фирмы Microsoft **за использование** Windows NT в качестве ОСРВ:

– Многопроцессность и многозадачность системы.
– Поддержка многопроцессорности.
– Preemption задач.
– Preemption прерываний и возможность их маскирования.
– Асинхронный ввод/вывод.
– Прямой доступ к оборудованию посредством интерфейса HAL (Hardware Abstraction Level). HAL обеспечивает изоляцию приложения от деталей реализации оборудования, обеспечивая платформенно-независимый прямой доступ к оборудованию.
– Специальная схема приоритетов. Все приоритеты разделены на две класса:

- Класс динамических приоритетов (0..15). Приоритеты динамически меняются планировщику по алгоритму близкому к принятому в Unix.
- Класс приоритетов реального времени (16, 22-26, 31). Эти приоритеты фиксированы, задачи из этого класса планируются на основе приоритетной очереди и получают управление раньше задач с динамическими приоритетами.

– Пространство ввода/вывода для задач из класса реального времени не участвует в страничном обмене механизма виртуальной памяти.
– Для закрепления страниц задачи в памяти существует специальный системный вызов (VirtualLock()).
– Представляются объекты синхронизации: критические секции, таймеры, события, mutex и др.

Аргументы **против использования** Windows NT в качестве ОСРВ:
– Ядро системы не является preemptive.
– Механизм DPC (Differed Procedure Call), вызываемый обработчиком прерываний, имеет недостатки:
  - Все процедуры DPC получают один и тот же приоритет и обрабатываются планировщиком в порядке поступления (FIFO). Тем самым низкоприоритетные прерывания будут обрабатываться ранее высокоприоритетных, но поступивших позднее.
  - Система не дает возможности узнать, сколько DPC стоит в очереди, поэтому нельзя узнать, когда начнет обрабатываться прерывание. Таким образом, существует случайная задержка между приходом прерывания и началом его обработки.

– Для каждого прерывания только один экземпляр DPC может быть в очереди. Следовательно процедура DPC должна уметь обрабатывать повторяющиеся прерывания, а оборудование должно уметь буферизовать прерывания во избежание потери данных. Это удорожает как драйверы устройств, так и оборудование.
– Малое количество приоритетов в классе реального времени (7 приоритетов) приводит к тому, что много задач будут иметь

одинаковый приоритет и планироваться алгоритмом типа round robin. Следовательно, время до начала исполнения задачи будет случайной величиной. Оно зависит от текущей загруженности системы.

– Не решена проблема инверсии приоритетов. Вместо традиционного для ОСРВ механизма наследования приоритетов Windows NT назначает задаче случайный уровень приоритета, позволяющий ей начать работу, что непредсказуемо и неприемлемо для ОСРВ. Поскольку приоритет задач класса реального времени не меняется, то этот механизм действует только на задачи из динамического класса. Тем самым ситуация только ухудшается: приоритет низкоприоритетных задач реального времени не меняется, а высокоприоритетные задачи могут быть вытеснены совсем низкоприоритетными задачами из динамического класса.
– Высокоприоритетные задачи могут блокироваться низкоприоритетными. Некоторые компоненты ядра работаю на уровне приоритета динамического класса. Следовательно, некоторые системные вызовы приводят к понижению приоритета и вывода задачи из класса реального времени. При этом она может быть блокирована другой задачей, имевшей до этого более низкий приоритет.
– Все страницы неактивного процесса, например, ожидающего данных, могут быть перенесены на диск, несмотря на закрепление их вызовом VirtualLock(). Это приводит к случайным задержкам при активации процесса (например, при поступлении ожидавшихся данных).
– Для Windows NT официально не приводятся времена системных вызовов и блокирования прерываний.
– Систему невозможно использовать без дисплея и клавиатуры.
– Систем предъявляет слишком большие запросы для ОСРВ на память.

Для устранения этих недостатков ряд компаний предлагает программные и аппаратные средства. Основные идеи их построения те же, что и для Linux. Поскольку исходный код системы недоступен, то, в отличие от Linux, существует только один способ приспособить систему к требованиям реального

времени: разработать микроядро, обеспечивающее надлежащее планирование задач и диспетчеризацию прерываний, а Windows NT выполнять как процесс.

Приведем несколько известных разработок такого рода.

Расширение RTX обеспечивает детерминированное управление системами на базе Windows XP, основными требованиями к которым являются стабильность рабочих характеристик и высокая степень масштабируемости; при этом RTX остается наиболее популярным расширением реального времени для Embedded Windows.

RTX глубоко интегрировано в ядро Windows XP и для обеспечения необходимых функций использует сервис Windows XP и API WIN32. Ядро реального времени (nucleus) интегрировано в ядро XP (kernel). Каждый процесс RTX выполняется как драйвер устройства ядра XP, при этом процессы не защищены друг от друга. Такая реализация приводит к быстрому переключению контекста, но небезопасна с точки зрения конфиденциальности.

Компания Ardence (VenturCom), благодаря своим расширениям реального времени, является лидером в области поддержки многоядерных технологий с 2002 года, когда начала разработку многопроцессорных конфигураций с двумя режимами функционирования.

В режиме с разделением задачи RTX исполняются в одном из ядер, в котором одновременно присутствуют и некоторые задачи Windows; при этом остальные ядра заняты исключительно исполнением задач Windows.

Режим с выделением обеспечивает реализацию преимуществ многоядерных технологий Intel. В такой конфигурации RTX-приложению полностью выделяется одно из ядер, а ОС Windows и менее критичные ко времени исполнения задачи загружаются в остальные ядра. Подобное решение существенно снижает уровень внутренних задержек тредов реального времени и предотвращает задержки исполнения тредов Windows XP, характерных для систем на одноядерных процессорах.

Программный интерфейс реального времени RTAPI (Real-Time API) является расширением Win32 и содержит, прежде

всего, набор функций, необходимых для управления устройствами. RTAPI реализован в двух видах – как подмножество вызовов подсистемы реального времени (RTSS — Real-Time Subsystem) и как динамическая библиотека (DLL), которая может вызываться из Win32-приложений. RTAPI содержит следующие группы функций:

− управление процессами и потоками — предоставляет Win32-совместимый интерфейс для управления, создания, изменения приоритетов, профилирования и завершения потоков реального времени,

− управление объектами RTSS — предоставляет возможности унифицированного управления объектами RTSS (создание, закрытие, доступ). Объектами RTSS являются: таймеры, обработчики прерываний и исключительных ситуаций (startup, shutdown, blue screen), потоки, процессы, семафоры, мьютексы, разделяемая память, почтовые ящики, консольный и файловый ввод-вывод, регистры.

Опыт компании Ardence, накопленный при разработке операционных систем реального времени (ОСРВ), позволил создать средство одновременного запуска менеджера виртуальной машины VMM (Virtual Machine Manager) и ядра ОСРВ. Оно будет поставляться в составе существующей версии **OCPB Phar Lap ETS** и обеспечит возможность исполнения разделенной управляемой виртуальной машиной тредов реального времени с жесткими детерминистичными характеристиками и ОС общего назначения Windows или Linux.

В RTX версии 7.0 появилась новая и полезная для разработчиков функция — поддержка отладки целевых систем средствами Visual Studio 2005 и .NET 2003. Это позволит выполнять отладку приложений RTSS, исполняющихся в целевой системе с исполнительными модулями RTX, из инструментальной системы.

В число других новых характеристик расширений RTX версии 7.0 входят:

− полнофункциональная поддержка Windows Server 2003 как в режиме с разделением, так и в режиме с выделением;

− поддержка Microsoft Visual Studio 2005;

– расширения данных отладчика RTX Debugger Data Extension с новыми командами, обеспечивающими более широкий доступ к внутренним структурам данных RTX;
– функция PerformanceView, благодаря которой разработчик получает истинное представление об общей загрузке системного процессора.

Расширения RTX широко используются в системах промышленной автоматизации, оборонных и аэрокосмических приложениях, тренажерах, робототехнике, контрольно-измерительной технике и медицинских приборах. Являясь программной средой на основе COTS-технологий, расширения RTX позволяют отказаться от применения дорогих и устаревающих частнофирменных аппаратных решений. Оценочные версии RTX можно загружать из Сети бесплатно. Дополнительную информацию о компании Ardence, расширении RTX 7.0 и других программных продуктах можно получить на сайте www.ardence.com

Система INtime является расширением реального времени Windows, которое было разработано корпорацией Radisys Corporation, а в настоящее время поддерживается корпорацией TenAsys

INtime комбинирует возможности ОСРВ жесткого реального времени со стандартными ОС Windows на платформе NT, не требуя дополнительной аппаратуры. INtime специально разработана под архитектуру процессора x86. Приложения реального времени и не реального времени выполняются на разных виртуальных машинах на единственном компьютере (рис. 5.1).

INtime, в отличие от RTX, слабо связана с ОС Windows NT. Архитектура INtime основана на механизме аппаратного обслуживания задач (hardware tasking), которое обеспечивается процессором Intel. Получается, что два ядра выполняются на одной аппаратуре. Поскольку они разделяют одну аппаратуру, потребовались некоторые модификации NT HAL. Такой подход позволяет защитить и отделить среду выполнения и область памяти от Windows. Внутри INtime каждый процесс приложения имеет свое собственное адресное пространство. Кроме того, яд-

ро и приложения выполняются на разных приоритетных уровнях, что позволяет защитить их друг от друга.

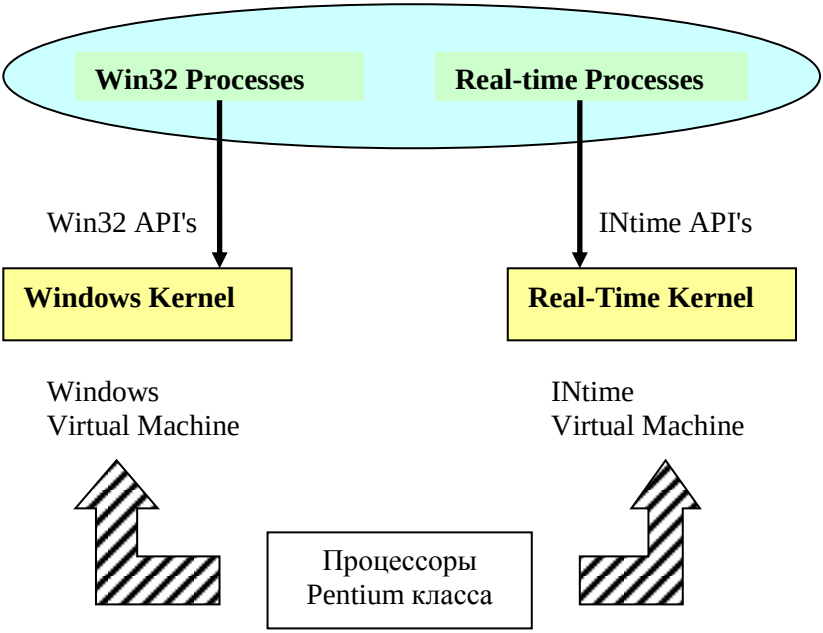

Рис. 5.1. Структура работы приложений в ОС INtime.

INtime показывает предсказуемое поведение, однако ее сложная архитектура не позволяет достичь системе хорошей производительности. Из-за сегментационных ограничений INtime подходит не для всех систем реального времени.

Операционные системы Microsoft Windows Embedded для встраиваемых систем имеют две разновидности в соответствии с версиями ОС Windows NT и Windows XP [MSEmb]. Версии систем Embedded корпорации Microsoft состоят из многочисленных конфигурируемых частей, которые позволяют легко манипулировать набором установленного программного обеспечения.

Windows NT Embedded использует технические ресурсы Windows NT и позволяет разрабатывать приложения, которые могут быть легко интегрированы в существующую информационную инфраструктуру (рис. 5.2).

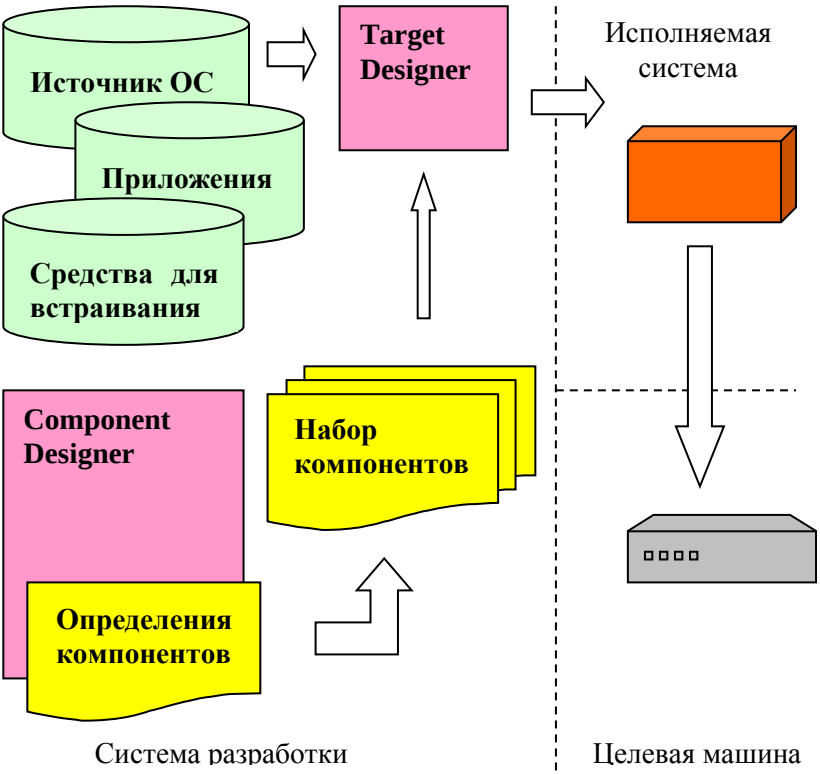

Рис. 5.2 Процесс разработки встраиваемого программного обеспечения на основе Windows NT Embedded

Набор средств разработки — Target Designer и Component Designer — позволяет OEM (original equipment manufacturer) производителям конфигурировать и создавать операционную систему для конкретной аппаратной платформы. Windows NT Embedded обладает специфическими компонентами для создания встраиваемых систем, которые позволяют работать в систе-

мах без видеоадаптера, осуществлять загрузку и работу накопителей в режиме «только чтение», выполнять удаленное администрирование и предоставляют дополнительные средства обработки ошибок и восстановления. Windows NT Embedded дает возможность создавать устройства, с которыми работать так же просто, как и со стандартными ПК на основе Windows, и управлять этими новыми устройствами на основе существующих профессиональных продуктов, таких как Microsoft Systems Management Сервер, HP OpenView, IBM Tivoli, CA Unicenter TNG, и др.

Разработчик встраиваемых систем применяет для конфигурирования ОС ***Target Designer***, используя готовый двоичный код Windows NT, дополнительные компоненты для встраивания и дополнительные приложения. В случае необходимости, для создания новых компонентов, не входящих в состав продукта (например, драйверов устройств, приложений и пр.), может использоваться ***Component Designer***. Вновь созданные новые компоненты могут быть импортированы в Target Designer и включены в состав целевой ОС. После конфигурирования ОС с помощью Target Designer происходит проверка взаимосвязей компонентов и строится образ системы, готовый к загрузке и исполнению на целевой системе.

Windows XP Embedded насчитывает до 10000 отдельных компонентов, а в Windows NT Embedded их было чуть больше 300. Основной отличительной чертой Windows XP Embedded является четкое разграничение компонентов системы, что позволяет разработчикам встраиваемого набора функций при создании образа системы включать только необходимые файлы и максимально сократить размер результирующей системы. Этими компонентами служат отдельные части системы Windows XP Professional.

Компоненты Windows XP Embedded представлены сервисами, приложениями, библиотеками и драйверами — разработчику нужно сконфигурировать необходимый набор функций и собрать из компонентов необходимую конфигурацию в образ среды исполнения (runtime image). Все опции конфигурации собраны воедино в базу данных компонентов. Разработчик имеет к

ней доступ и может ее редактировать с помощью специального инструмента — Component Database Manager.

Для каждого компонента в процессе создания определяется ряд параметров:

– платформа, на которой будет выполняться данный компонент (определяет порядок компиляции и сборки);
– описание и схема подключения компонента;
– список ассоциированных ресурсов, таких как файлы и ключи реестра;
– зависимости компонента от других компонентов (например, от DirectX или NET runtime);
– указатель на хранилище файлов (чаще всего это просто локальный каталог, но может быть и сетевым ресурсом);
– принадлежность к группе для упрощения обращения сразу к нескольким компонентам как к целому.

Сама база данных управляется СУБД MS SQL Server и может быть расположена как локально, на компьютере разработчика, так и на сервере.

Realtime ETS Kernel. Система Realtime ETS Kernel выпускается фирмой Phar Lap SoftWare в двух вариантах:

1. TNT Embedded ToolSuite, Realtime Edition, включающий: Realtime ETS Kernel, ETS TCP/IP, отладчик CodeView с поддержкой Borland Turbo Debugger, ассемблер 386ASM, linker, поддержку компиляторов Visual C/C++, Borland C/C++, Watcom C/C++ и API Win32.

2. Realtime ETS Kernel — полная замена Windows NT, включает компактное ядро (28Кб), поддерживающее Win32 API и использующее стандартные библиотеки компиляторов. Ядро имеет 32 уровня приоритетов и может быть записано в ПЗУ.

Hyperkernel. Система Hyperkernel выпускается фирмой Nematron (Imagination Systems). Представляет собой ядро, обеспечивающее детерминированное планирование и работающее на уровне привилегий 0 процессора Intel 80x86 вместе с Windows NT. Задачи HyperKernel не видны Windows NT. Для них определены 8 уровней приоритетов с preemptive планированием. В ка-

честве средства разработки используются стандартные для Windows NT компиляторы Visual C/C++ и специальные библиотеки. Используются API Win32 и стандартный HAL. Разрешение таймера: 1 микросекунда, минимальный квант времени 20 микросекунд. Время задержки на прерывание — 5микросекунд, переключение контекста — 4 микросекунды на Intel Pentium 133МГц.

5.3. Собственно типы ОСРВ

5.3.1. LynxOS

Система LynxOS выпускается фирмой LynuxWorks (USA).

Основные характеристики:

Категория: self-hosted.

Архитектура: на основе микроядра.

Стандарт: собственный и POSIX 1003.

Процессоры (target): Intel 80x86, Motorola 68xxx, SPARC, PowerPC.

Размер ядра: 33Кб.

Средства синхронизации и взаимодействия: POSIX 1003.

Планирование: приоритетное, FIFO, Round Robin, preemptible.

Средства разработки:

Комплект разработки C/C++, включающая компиляторы, отладчик, анализатор.

X Window/Motif для Lynx.

Total View — многопроцессный отладчик.

В ОС LynxOS были впервые реализованы многие идеи, впоследствии подхваченные другими поставщиками операционных систем реального времени. Например, согласованность со стандартом POSIX или поддержка Linux-приложений.

Компания LynuxWorks была основана в 1988 и вышла на рынок с первой в мире реализацией UNIX реального времени: операционной системой LynxOS. Первоначально компания называлась Lynx Real-Time Systems. В мае 2000 года название компании изменилось на LynuxWorks, став отражением новой стратегии, воплощенной в формуле:

Lynx + Linux = Lynux

С одной стороны, эта формула демонстрировала факт разработки компанией своей собственной версии продукта класса Embedded Linux (BlueCat Linux), с другой стороны, стремление в самой ОСРВ LynxOS обеспечить определенный уровень совместимости с Linux, который впоследствии воплотился в так называемый уровень двоичной совместимости с Linux: Linux ABI (Application Binary Interface).

Первая версия LynxOS была написана в 1986 году в Далласе (Техас) для процессора Motorola 68010. В 1988-89 году LynxOS была портирована на Intel 80386.

LynxOS 2.0 (февраль 1991 г.). Поддерживала относительно небольшой набор возможностей, таких как двоичные семафоры, разделяемая память, асинхронные события, межпроцессное взаимодействие, непрерывные файлы. В конце 1994 года LynxOS стала поддерживать процессоры PowerPC 60x, что явилось предпосылкой к значительному расширению рынка использования LynxOS и, в конечном счете, к завоеванию лидирующих позиций LynxOS среди других ОСРВ именно в сегменте оборудования на платформе PowerPC. С сентября 1995 года были начаты работы по портации LynxOS на процессоры SPARC-архитектуры.

LynxOS 3.0 (март 1998 г.). Поддерживается широкий спектр процессорных архитектур: x86 (Intel 386, 486, Pentium, Pentium Pro, Pentium II), PowerPC (603, 603e, 603ev, 604, 604e, 604er, PowerQUICC 860,821, 750), 68K (MC68030, MC68040, MC68LC040, MC68060), SPARC (microSPARC, microSPARC II).

LynxOS 4.0 (июнь 2000 г.). Основные целевые платформы: x86, PowerPC, MIPS (специальная версия для HP). Кросс-платформы Windows 2K, XP, Linux RedHat 7.2 и выше, Solaris 2.7 и выше. Поддержка Linux ABI для ядра Linux 2.4.x. Расширение сетевых возможностей, в том числе поддержка IPv6 и IPSec.

LynxOS 5.0 — 2008 год. Поддержка симметричной мультипроцессорной обработки данных: выпуск новой версии был обусловлен необходимостью получения разработчиками адекватного продукта, соответствующего современному уровню развития процессорных технологий.

Основной отличительной особенностью версии ОС LynxOS 5.0 является поддержка многоядерных конфигураций и архитектуры симметричной мультипроцессорной обработки данных (SMP). Надо сказать, что в ОС LynxOS поддержка многоядерных конфигураций была в старых версиях LynxOS вплоть до версии 2.5, но потом перестала поддерживаться.

Приведем важные пользовательские отличия ОС LynxOS 5.0 от версий LynxOS 4.0/4.2:

− обновленный набор средств разработки GNU toolchain, основанный на GCC, G++ версии 3.4.3 и GDB версии 6.5;

− в ОС LynxOS 5.0 разработка приложений и драйверов теперь доступна только в кросс-среде (Linux или Windows), в том время как в предыдущих версиях (4.0/4.2) дополнительно поддерживалась native среда разработки (то есть среда разработки в самой LynxOS);

− в ОС LynxOS 5.0 все библиотеки поставляются только в виде многопотоковых версий, в том время как в предыдущих версиях (4.0/4.2) поддерживались как однопотоковые, так и многопотоковые версии библиотек. Это обеспечивает универсальность разрабатываемого пользователем программного обеспечения;

− поддержка файловой системы FAT16/32, что позволяет использовать файлы из-под DOS и Windows;

− все файлы в системе стали теперь в ELF формате. В предыдущих версиях, часть файлов была в формате coff (например, ядро), часть в формате elf (например, приложения пользователя), что создавало определенные неудобства для пользователя; поддержка языка Ada (GNAT 3.4.3);

− полная поддержка многопотоковой версии C++;

− новый, более современный уровень согласованности с POSIX: POSIX—POSIX 1003.1-2003 PSE 53/54;

− двоичная ABI совместимость с Linux 2.6. В предыдущих версиях ОС LynxOS (4.0/4.2) поддерживалась с Linux 2.4. Новая версия Linux ABI позволяет запускать в среде LynxOS 5.0 такие серьезные приложения как Oracle 9, Apache, ACE-TAO, PostGres и QT;

− новейшая интегрированная среда разработки Luminosity 3.0, в основе которой лежит популярная инструментальная платформа Eclipse. Luminosity представляет собой полнофункцио-

нальную интегрированную среду разработки на базе Java, предназначенную для использования в системах LynxOS для создания, редактирования, компилирования, контроля и отладки встраиваемых приложений и программ реального времени. А также возможность использования ОС LynxOS 5.0 была обеспечена для аппаратной платформы x86 (процессоров Intel и AMD), а также для нескольких плат с процессором PowerPC (MPC74xx, MPC8640/8641D) от компаний GE Fanuc Intelligent Platforms, Curtiss-Wright и Freescale Semiconductor.

Функциональные возможности ОС **LynxOS 5.0**

Поддержка SMP-архитектуры. ОС LynxOS 5.0 может работать в режиме SMP на платах с несколькими процессорными ядрами или процессорами. Чтобы обеспечить поддержку SMP в ОС LynxOS 5.0, было сделано несколько изменений в ядре операционной системы и библиотеках пользовательского уровня. Эти изменения оказывают определенное влияние на процесс разработки заказных ядер операционной системы LynxOS и приложений пользователя.

Главной особенностью систем с архитектурой SMP является наличие общей физической памяти, разделяемой всеми процессорами. При обращении к памяти все процессоры имеют равные права и одну и ту же адресацию для всех ячеек памяти. Поэтому SMP-архитектура называется симметричной.

Существует два подхода к организации потоков очередей на выполнение в среде SMP:

− первый подход — на каждый процессор формируется своя отдельная очередь потоков на выполнение. Другими словами, если, например, в системе есть два процессора (ядра), то к каждому процессору будет организовываться своя отдельная очередь (то есть будет две очереди потоков).

− второй подход — для всех процессоров формируется одна общая очередь потоков на выполнение. Планировщик определяет, на каком из процессоров будет выполняться поток в данный момент.

Планирование в ОС LynxOS. Будучи операционной системой жесткого реального времени с планировщиком со строгим вытеснением по приоритетам, ОС LynxOS рассматривает время отклика в реальном времени как более важный параметр,

чем суммарная производительность системы. Поэтому был выбран подход с ***одной очередью на выполнение***. В ОС LynxOS 5.0 существует только одна очередь потоков на выполнение в системе (или в разделе, если версия LynxOS обеспечивает поддержку изолированных разделов, например, как в LynxOS-178). Когда поток становится готовым к работе, планировщик немедленно запускает его на одном из процессоров с самым низким текущим приоритетом выполняемого потока. Этот подход гарантирует, что в системе с N процессорами в любой момент времени выполняются N потоков с самым высоким приоритетом. С другой стороны, перемещение потоков между процессорами происходит более часто, чем в подходе с множеством очередей на выполнение, и таким образом приводит к сокращению суммарной производительности системы.

Многие современные многоядерные архитектуры подвержены этому эффекту намного меньше, т.к. имеют в своем составе общий для всех ядер кэш 2-го или 3-го уровня, значительно ускоряющий перемещение контекста потока между ядрами.

Механизмы синхронизация в ОС LynxOS 5.0 на уровне ядра.

В ОС LynxOS 5.0 для обеспечения работы в SMP среде были добавлены новые механизмы синхронизации на уровне ядра операционной системы: атомарные операции, спин-блокировки (spin lock) ядра, большая блокировка ядра (BKL — Big Kernel Lock).

Новые механизмы синхронизации имеют аналоги в Linux. Новые механизмы дополнили список механизмов синхронизации на уровне ядра, использовавшиеся для однопроцессорной архитектуры: семафоры ядра, запрет прерывания (на вызывающем процессоре), запрет вытеснения потока (на вызывающем процессоре).

Спин-блокировки (взаимные блокировки) предоставляют особый способ обеспечения взаимных исключений посредством циклов активного ожидания блокировок. Если блокировка доступна, она захватывается, взаимно исключающее действие выполняется, и блокировка снимается. Если блокировка недоступна, поток переводится в состояние активного ожидания блокировки, пока она не освободится. Спин-блокировка используется

в критических секциях (части кода, защищенной от одновременного доступа) ядра операционной системы и влечет за собой запрет прерывания и вытеснения потоков на конкретном процессоре. При этом на другом процессоре задачи могут продолжать выполняться в обычном режиме (без запрета на прерывания и вытеснение).

BKL — специальный вид спин-блокировки, который существует только в единственном экземпляре на всю систему. BKL может использоваться только одной задачей в системе в любой момент времени. В отличие от обычной спин-блокировки, BKL влечет запрет прерывания и вытеснения потоков на всех процессорах в системе.

Атомарные переменные и операции. ОС LynxOS 5.0 поддерживает атомарный тип переменных, который является значением целого числа, хранимого в ОЗУ, плюс ряд поддержанных аппаратными средствами операций для атомарных переменных (чтение, изменение, запись, тестирование и установка). Атомарные операции на атомарных переменных являются непрерываемыми для всей системы, и поэтому полезны для синхронизации и других целей и в однопроцессорной и многопроцессорной среде.

Рассмотрим на примере подсчета ссылок, для чего нужны атомарные переменные и операции над ними в SMP среде. Пусть процесс хочет отказаться от своей доли в разделяемом ресурсе и узнать, владеет ли им кто-то еще, путем уменьшения счетчика этого разделяемого ресурса и проверки его на равенство нулю. Типичная последовательность действий может начинаться со следующего:

1. Процессор загружает текущее значение счетчика, например, 2, в один из своих регистров.
2. Процессор уменьшает это значение в своем регистре; теперь оно равно 1.
3. Процессор записывает новое значение (1) обратно в память.
4. Процессор решает, что поскольку значение равно 1, разделяемый объект используется каким-то другим процессом, поэтому не освобождает объект.

В однопроцессорных системах этот сценарий не вызывает проблем. Но в симметричных мультипроцессорных системах

картина совершенно иная: а что будет, если окажется, что другой процессор выполняет ту же работу и в то же время? Наихудший случай выглядит примерно так:

1. Процессор А загружает текущее число 2 в один из своих регистров.
2. Процессор В загружает текущее число 2 в один из своих регистров.
3. Процессор А уменьшает значение в своем регистре; теперь оно равно 1.
4. Процессор В уменьшает значение в своем регистре; теперь оно равно 1.
5. Процессор А записывает новое значение (1) обратно в память.
6. Процессор В записывает новое значение (1) обратно в память.
7. Процессор А решает, что поскольку значение равно 1, этот разделяемый объект используется каким-то другим процессом, поэтому не освобождает его.
8. Процессор В решает, что поскольку значение равно 1, этот разделяемый объект использует какой-то другой процесс, поэтому не освобождает его.

Число ссылок в памяти теперь должно быть равно 0, но вместо этого, оно равно 1. Оба процессора удалили свои ссылки на разделяемый объект, но ни один из них его не освободил. Каждый процессор выполнил именно то, что требовалось, и все равно возник неправильный результат. Проблема, безусловно, состоит в том, что процессоры не синхронизировали свои действия. Простейшим средством синхронизации ядра ОС LynxOS в SMP-среде являются атомарные операции.

ОС LynxOS 5.0 обеспечивает многие операции, которые могут быть атомарно выполнены над значением целого числа, расположенного в ОЗУ. ***Атомарно означает, что операция появляется на всех процессорах и потоках в системе как единственный непрерываемый доступ памяти.*** Атомарные операции являются самыми быстрыми (по сравнению с другими методами синхронизации) доступными методами синхронизации в случае, когда для синхронизации доступа требуется только одно целое число, поскольку они позволяют пользователю

избегать более тяжелых примитивов синхронизации, таких как семафоры и спин-блокировки. Однако атомарные операции обычно намного более медленные, чем соответствующие неатомарные операции. API для атомарных операций определено в заголовочном файле *atomic_ops.h*. Атомарные операции с целыми числами доступны как при работе ядра, так и в приложениях пользователя.

Все атомарные операции воздействуют на платформенно зависимую переменную типа atomic_t (называемую атомарной переменной). Целое значение числа, содержащееся в типе atomic_t, имеет тип atomic_value_t. Это 32-битовое число со знаком. Тип atomic_t преднамеренно определен так, чтобы переменные этого типа не могли применяться в выражениях на языке C с целыми числами, а только использовались с помощью API для атомарных операций.

Вызов atomic_init (av, v) инициализирует атомарную переменную av и устанавливает ее в указанное значение v. Все атомарные переменные должны инициализироваться прежде, чем они смогут использоваться. Другой способ инициализировать атомарную переменную состоит в том, чтобы присвоить ей начальное значение ATOMIC_INITIALIZER (v) при определении переменной, где v — ее начальное значение:

```
#include <atomic_ops.h>
…
atomic_t av1 = ATOMIC_INITIALIZER (0);
atomic_t av2;
…
atomic_init (av2, 0);
```

Операции над атомарными переменными приведены в таблице 5.1.

Таблица 5.1. Операции над атомарными переменными в ОС LynxOS

Операция	**Описание**
atomic_set(av, v)	Установить атомарную переменную в указанное значение
atomic_get(av)	Возвратить текущее значение атомарной переменной
atomic_add(av, v)	Добавить указанное значение к атомарной переменной и возвратить ее новое значение
atomic_sub(av, v)	Вычесть указанное значение из атомарной переменной и возвратить ее новое значение
atomic_inc(av)	Увеличить на 1 значение атомарной переменной и возвратить ее новое значение
atomic_dec(av)	Уменьшить на 1 значение атомарной переменной и возвратить ее новое значение
atomic_xchg(av, v)	Установить значение атомарной переменной av в v и возвратить ее старое значение
atomic_cmpxchg(av,v1,v2)	Взять два значения, v1 и v2. Если значение атомарной переменной равно v1, то установить ее в v2. В любом случае возвращается старое значение атомарной переменной
atomic_or(av, v)	Сделать побитовое ИЛИ значения атомарной переменная с указанной битовой маской и сохранить результат в атомарной переменной. Никакое значение не возвращается

atomic_and(av, v)	Сделать побитовое И значения атомарной переменной с указанной битовой маской и сохранить результат в атомарной переменной. Никакое значение не возвращается
atomic_xor(av, v)	Сделать побитовое XOR значения атомарной переменной с указанной битовой маской и сохранить результат в атомарной переменной. Никакое значение не возвращается
atomic_get_or(av, v)	Сделать побитовое ИЛИ значения атомарной переменной с указанной битовой маской и сохранить результат в атомарной переменной. Возвращается старое значение
atomic_get_and(av, v)	Сделать побитовое И значения атомарной переменной с указанной битовой маской и сохранить результат в атомарной переменной. Возвращается старое значение

Барьеры памяти

Синхронизация доступа к разделяемому ресурсу требует команд доступа к памяти (сохранения и загрузки) для того, чтобы выполняться и стать видимым для всех процессоров в системе в определенном порядке. Обычно программист ожидает, что доступ к памяти будет выполнен в том же самом порядке, в котором он появляется в коде программы. Однако, без дополнительных уловок, о которых будет написано ниже, это может оказаться неправильным, потому что современные оптимизирующие компиляторы могут переупорядочить инструкции процессора (в общем случае) и доступ к памяти (в частности) для того, чтобы достигнуть лучшей производительности. Например: если

программа сохраняет 1, а затем 0 в то же самое место памяти, то компилятор может исключить первое сохранение. Однако, если первое сохранение было ***блокированием ресурса,*** а второе было ***разблокированием ресурса,*** то оптимизация компилятора приведет к неправильной работе программы.

Современные процессоры порождают другую проблему: большинство современных архитектур процессоров могут выполнять программу не в том порядке, как это написано в программе, а в порядке того, как данные становятся доступными (например, чтение из кэшируемой области памяти может быть выполнено прежде, чем будет выполнено предшествующее чтение из некэшируемой области памяти). У каждой архитектуры процессора есть ряд правил для упорядочения выполнения доступа к памяти, который называется моделью памяти. Во всех архитектурах отдельный процессор является последовательным, что заставляет программу, работающую на этом процессоре, выглядеть так, что все ее инструкции выполняются в порядке, указанном в программе. Однако это становится проблемой, когда есть другие «пользователи» на шине памяти, такие как внешние устройства ввода/вывода и, что наиболее важно, другие процессоры.

Как уже сказано, переупорядочение доступа памяти компилятором и процессором может потребовать синхронизации доступа к ресурсу. Рассмотрим общую последовательность операций в критической области кода, защищающего разделяемый ресурс:

1. Блокировка ресурса.
2. Доступ к ресурсу.
3. Разблокировка ресурса.

Все три пункта содержат доступы памяти. Крайне важно, чтобы пункт 1 был выполнен сначала, затем пункт 2, затем пункт 3. Однако, это, возможно, не настолько очевидно для оптимизирующего компилятора и процессора, которые видят только последовательность команд, не зная их цель. Если компилятор или процессор переставляют команды доступа к памяти пункта 2 с пунктом 1 или пунктом 3, то эти команды выпадают из критической области, чего определенно не хочет программист.

ОС LynxOS 5.0 реализует ряд вызовов, чтобы **обеспечить желательный порядок доступа к памяти — *барьеры памяти***. Есть два вида барьеров памяти: общий и специализированный.

Общие барьеры памяти гарантируют выполнение своих функций в любой ситуации; однако, они являются обычно весьма медленными. В определенных случаях могут использоваться менее универсальные барьеры, которые называются ***специализированными барьерами***.

В общем случае, барьер памяти позволяет осуществлять выполнение доступа к памяти определенного типа в том же порядке, в каком они появляются в программе.

API барьеров памяти определен в заголовочном файле membar.h. Барьеры памяти доступны и в ядре и в программах пользователя. Общие барьеры памяти приведены в таблице 5.2.

Таблица 5.2. Описание барьеров памяти в ОС LynxOS

Барьер памяти	**Описание**
membar_compiler ()	Барьер оптимизации компилятора. Защищает компилятор от переупорядочения порядка операций доступа к памяти через этот барьер в целях оптимизации. Это также заставляет компилятор прекратить все предположения о содержимом памяти. Это — основной барьер: все другие барьеры включают функциональные возможности барьера оптимизации компилятора
membar_store ()	Барьер сохранения. Гарантирует, что все сохранения, предшествующие этому барьеру (в порядке, указанном в программе), выполнены и глобально видимы перед сохранениями, следующими после этого барьера
membar_load ()	Барьер загрузки. Гарантирует, что все загрузки, предшествующие этому барьеру (в порядке, указанном в программе), выполнены перед загрузками, следую-

	щими после этого барьера
membar_all ()	Полный барьер памяти. Гарантирует, что все доступы к памяти (загрузки и сохранения), предшествовавшие этому барьеру (в порядке, указанном в программе) выполнены и глобально видимы перед доступами к памяти, следующими после этого барьера

Особенности генерация ядра ОС LynxOS 5.0 для работы с несколькими процессорами (ядрами)**.**

Каждый процессор может выполнять один поток в каждый момент времени. Следовательно, система с N процессорами в состоянии выполнять одновременно N потоков.

При генерации ядра ОС LynxOS 5.0 для работы с несколькими процессорами надо задать значение параметра NUM_CPUS в файле

$ENV_PREFIX/sys/bsp.<bsp_name>/uparam.h

Если параметра NUM_CPUS нет в файле uparam.h, то ОС LynxOS не будет поддерживать SMP и будет использовать только один процессор. Если параметр NUM_CPUS определен и равен 0 (по умолчанию), то ОС LynxOS будет стараться определить число процессоров автоматически и использовать все найденные процессоры.

Обычно этого достаточно для большинства конфигураций, но иногда желательно явно указать число процессоров, например, для того чтобы уменьшить время загрузки операционной системы. Если параметр NUM_CPUS равен –1, то число процессоров берется из установок BIOS. Если параметр NUM_CPUS равен 1, то ОС LynxOS работает в однопроцессорной конфигурации. И, наконец, если параметр NUM_CPUS не меньше двух, то он задает число процессоров, которые должны использоваться операционной системой.

Основными аппаратными платформами, которые поддерживаются всеми версиями ОС LynxOS, являются x86 и PowerPC. Кроме того, существуют специальные заказные версии для процессоров MIPS. Кроме ОС LynxOS компания

LynuxWorks предлагает пользователям еще четыре системных продукта: LynxOS-178, LynxOS-SE, LynxSecure, BlueCat Linux.

5.3.2. OS-9

Система OS-9 выпускается фирмой Microware (USA).

Основные характеристики:

Категория: host/target.

Архитектура: на основе микроядра.

Стандарт: собственный.

ОС разработки (host): Unix/Windows

Процессоры (target): Intel 80x86, Motorola 68xxx, ARM, PowerPC, MIPS и др.

Линии связи host/target: последовательный канал и Ethernet.

Размер ядра: 16Кб.

Средства синхронизации и взаимодействия: разделяемая память, сигналы, семафоры, события и т.д.

Планирование: приоритетное, FIFO, специальный механизм планирования, preemptible.

Средства разработки:

Hawk — интегрированная среда разработки на C/C++.

PersonalJava — виртуальная машина Java.

В 1979 году совместными усилиями фирм Microware и Motorola была разработана операционная система реального времени для микропроцессора 6809. Версия Level I OS-9/6809 была способна адресовать 64 KB памяти. Версия Level II OS-9/6809, используя прогрессивные в то время аппаратные средства динамической трансляции адресов, была способна адресовать уже до 2 MB памяти и стала применяться в ряде популярных компьютерных систем, например таких как, Tandy Color Computer III.

В 1982 году Microware (уже независимо от Motorola) портировала ОС OS-9 для семейства микропроцессоров 68000, создав систему OS-9/680X0 для 16-ти и 32-х разрядных микропроцессоров и микроконтроллеров. Код системы лишь на 20% был написан на языке высокого уровня (фрагмент распределения памяти и настраиваемая часть загрузочного кода), остальная

часть с целью достижения максимальной производительности написана на Ассемблере. За пять лет с момента появления ОС OS-9/680X0 стала признанным промышленным стандартом дефакто для операционных систем реального времени (и абсолютным лидером по применимости в промышленных приложениях на базе технологии VME).

В списке поддерживаемых микропроцессоров ОС OS-9/68K наиболее полно представлено семейство 68К — от младшего в серии MC68000 до 32-х разрядного, суперскалярного MC68060. Наибольшую популярность и распространение получили версии 2.4/2.5 системы и в настоящее время пользователями по достоинству оценены пользователями мощь и надежность новейшей версии системы — 3.0.1.

ОС OS-9000 — переносимая версия ОС OS-9, написанная главным образом (95%) на С. Оставшиеся, критичные с точки зрения производительности участки кода, написаны на Ассемблере. Как результат, только 5% ОС OS-9000 необходимо переписать, чтобы перенести ее на новый процессор. Теоретически ОС OS-9000 может быть перенесена на любую современную микропроцессорную архитектуру.

Для пользователя ОС OS-9 и ОС OS-9000 — UNIX подобная среда. Модель процессов, межпроцессная коммуникация, многопользовательская файловая система и большое количество стандартных для UNIX утилит — все это позволит программисту знакомому с UNIX в минимально короткий срок почувствовать себя уверенно в среде реального времени ОС OS-9. Единство архитектурных решений, совместимость приложений на уровне исполняемого кода, технология реализации интегральных средств разработки, а также дизайна и состава системных продуктов — эти факторы объединяют операционные системы реального времени OS9 и OS9000 в семейство операционных систем OS9 (рис. 5.3).

Система OS-9 является модульной, чрезвычайно гибко конфигурируемой, высокопроизводительной, встраиваемой системой реального времени. Именно такой она задумывалась с самого начала и именно таким видится путь ее дальнейшего развития.

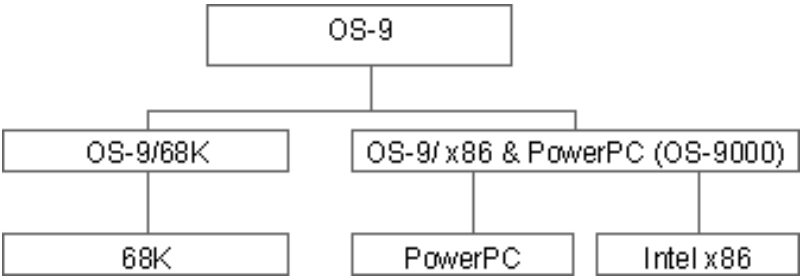

Рис. 5.3. Семейство операционных систем OS9

Совокупность требований, исходя из которых проектировалась ОС OS-9, в самом общем случае сводится к обеспечению следующих потребительских свойств создаваемой коммерческой ОС РВ:

- минимизация объема (компактность);
- гибкость (модифицируемость);
- возможность наращивания функциональности;
- доступность по цене (и в первую очередь для выпускаемой массово продукции);
- охват наиболее широкого спектра задач/приложений реального времени.

Если первые четыре пункта вполне интуитивно понятны, то последний требует более полной характеристики.

ОС OS-9 предоставляет профессиональный набор программных системных средств реального времени, позволяющий в наиболее широком спектре приложений найти оптимальное сочетание производительности и функциональности средств уровня операционной системы для поддержания высокой надежности и производительности целевой системы РВ.

Ядро ОС OS-9 обеспечивает: диспетчеризацию процессов на основе приоритетов с (необязательным) разделением времени, обслуживание прерываний, обмен информацией между процессами, обработку ошибок, распределение и защиту всех разделяемых системных ресурсов.

При реализации каждой из указанных функций предпринимались все меры обеспечения максимальной степени надежности. К примеру, ядро управляет всеми разделяемыми ресур-

сами (получая и отвечая на запросы исполняющихся процессов), вместо того, чтобы просто позволить различным процессам конкурировать за разделяемый ресурс. При таком подходе гарантируется, что допущенная программистом ошибка при создании прикладной программы не приведет к краху всей системы. Кроме того, защита выделенной ядром исполняющемуся процессу памяти выполняется диспетчером памяти.

В системе имеются следующие требующие защиты ресурсы:

- содержимое памяти,
- регистры портов ввода/вывода,
- регистры системных часов,
- процедуры перехвата сигналов в процессах,
- процедуры обработки ошибок,
- интервалы приостановки исполнения,
- приоритеты диспетчеризации,
- пользовательские и групповые привилегии.

Изменение любых из этих величин, выполняемое в процессе работы системы, должно выполняться очень осторожно и тщательно во избежание возможного краха системы. Единственный способ гарантии того, что все изменения будут производиться с соблюдением необходимых мер предосторожности — это поручить ядру задачи распределения и защиты этих ресурсов. При прекращении исполнения процесса (нормальным образом или аварийно) ядро определяет, какие ресурсы были связаны с этим процессом в момент прекращения выполнения и освобождает эти ресурсы.

Эффективность обслуживания прерываний. Прерывания в ОС OS-9 обрабатываются соответствующими подпрограммами, которые могут быть установлены в операционной системе либо драйвером устройства, либо прикладным процессом.

Базовые файловые менеджеры ОС OS-9 предназначены для обеспечения обмена информацией между процессами и обеспечивают приложениям ОС OS-9 доступ к различным последовательным устройствам типа принтеров и терминалов, а также к устройствам внешней памяти типа дисков (жестких, гибких, электронных и оптических) и лент.

Модульная структура ОС OS-9 позволяет системному разработчику выбирать именно те функциональные блоки, которые требуются данному приложению. Для обеспечения локального хранения и регистрации данных в систему может быть включена поддержка дисковых и ленточных устройств. Сетевые файловые менеджеры обеспечивают доступ к самым разным сетевым устройствам по протоколу TCP/IP и прозрачно связывать между собой различные системы. Любая из опций ОС OS-9 легко может быть добавлена в систему для поддержки изменившихся требований к системе.

5.3.3. VxWorks

Система VxWorks выпускается фирмой Wind River Systems (USA).

Основные характеристики:

Категория: host/target.

Архитектура: монолитная.

Стандарт: собственный и POSIX 1003.

ОС разработки (host): Unix/Windows

Процессоры (target): Intel 80x86, Motorola 68xxx, Intel 80960, ARM, PowerPC, MIPS, Alpha, SPARC.

Линии связи host/target: последовательный канал, Ethernet, шина VME.

Размер ядра: 5-8Кб.

Средства синхронизации и взаимодействия: POSIX 1003.

Планирование: приоритетное, preemptible.

Средства разработки:

Tornado — интегрированная среда разработки на C/C++.

VxSim — эмулятор для Unix.

WindView — графический визуализатор состояния задач.

Название ***VxWorks*** получилось из игры слов с названием ***ОСРВ VRTX***[2], созданной компанией Ready Systems. В начале

[2] **Система VRTX** (versatile real-time executive — универсальный диспетчер реального времени) выпускается фирмой Ready Systems (USA).

Основные характеристики:

Категория: host/target.

восьмидесятых VRTX была достаточно новым и сырым продуктом и ее нельзя было использовать как полноценную операционную систему. Компания Wind River приобрела права на распространение расширенной версии VRTX под названием VxWorks.

Доработки и расширения, внесённые компанией WindRiver, позволили создать систему, которая работала (например, VXWorks имела файловую систему и интегрированную среду разработки) таким образом. Название VxWorks может означать VRTX now Works («VRTX теперь работает») или VRTX that Works}} («VRTX, которая работает»).

Когда стало ясно, что Ready Systems может разорвать контракт на распространение VRTX, в Wind River было разработано собственное ядро операционной системы, которое заменило VRTX. Базовая функциональность нового ядра VxWorks была такой же как у VRTX.

Операционная система VxWorks является системой с кросс-средствами разработки прикладного программного обеспечения, разработка ведется на инструментальном компьютере (host) в среде Tornado для последующего исполнения на целевой машине (target) под управлением ОС VxWorks. Данная ОС имеет большое количество версий для различных сфер деятельности.

Стандарт: собственный.

ОС разработки (host): Unix/Windows

Процессоры (target): Intel 80x86, Motorola 68xxx, Intel 80960, PowerPC.

Линии связи host/target: последовательный канал, Ethernet, шина VME.

Размер ядра: 16Кб.

Средства синхронизации и взаимодействия: семафоры, события, mutex и д.р.

Планирование: приоритетное, preemptible.

Средства разработки:

MasterWorks — интегрированная среда разработки.

Xray — специализированный отладчик.

Simulator Xray — эмулятор ядра.

Операционная система VxWorks построена, как и положено ОС жесткого реального времени, по технологии микроядра, т. е. на нижнем непрерываемом уровне ядра выполняются только базовые функции планирования задач и их управления коммуникацией/синхронизацией.

Все остальные функции операционной системы более высокого уровня (управление памятью, вводом/выводом, сетевые средства и т. д.) базируются на простых функциях нижнего уровня, что позволяет обеспечить быстродействие и детерминированность ядра, а также легко строить необходимую конфигурацию операционной системы.

В многозадачном ядре ***wind*** применен алгоритм планирования задач, учитывающий приоритеты и включающийся по прерываниям. В качестве основного средства синхронизации задач и взаимоисключающего доступа к общим ресурсам в ядре wind применены семафоры. Имеется несколько видов семафоров, ориентированных на различные прикладные задачи: двоичные, целочисленные, взаимного исключения и POSIX.

Все аппаратно-зависимые части ОС VxWorks вынесены в отдельные модули для того, чтобы разработчик встроенной компьютерной системы мог сам переставить ОС VxWorks на свою нестандартную целевую машину. Этот комплект конфигурационных и инициализационных модулей называется (Board Support Package, BSP) и поставляется для стандартных компьютеров (VME-процессор, PC или Sparcstation) в исходных текстах. Разработчик нестандартной машины может взять за образец BSP наиболее близкий по архитектуре стандартный компьютер и перенести ОС VxWorks на свою машину путем разработки собственного BSP с помощью BSP Porting Kit.

Базовые сетевые средства ОС VxWorks: UNIX-networking, SNMP и STREAMS.

ОС VxWorks была первой операционной системой реального времени, в которой реализован протокол TCP/IP с учетом требований реального времени. С тех пор ***ОС VxWorks поддерживает все сетевые средства, стандартные для UNIX***:

- TCP/UDP/ICMP/IP/ARP,
- Sockets, SLIP/CSLIP/PPP,
- telnet/rlogin/rpc/rsh, ftp/tftp/bootp,

− NFS (клиент и сервер).

Реализация SNMP-агента с поддержкой как MIB-I, так и MIB-II предназначена для применения ОС VxWorks в интеллектуальном сетевом оборудовании (хабы, мосты, маршрутизаторы, повторители) и других устройствах, работающих в сети.

STREAMS — стандартный интерфейс для подключения переносимых сетевых протоколов к операционным системам, реализован в ОС VxWorks как в версии SVR3, так и SVR4. Таким образом, в ОС VxWorks можно инсталлировать любой протокол, имеющий STREAMS-реализацию, как стандартный (Novell IPX/SPX, DECNET, Apple-Talk и пр.), так и специализированный.

В ОС VxWorks поддерживаются следующие файловые системы:

− MS-DOS-Compatible File System: DosFS;
− Raw File System: RawFs;
− Target Server File System: TSFS;
− Network File System: NFS;
− ISO 9660 (CDROM File System);
− Tape File System: TapeFs;
− CIFS/SMB;
− TrueFFS.

Wind River Systems анонсировала (1994) программу WindNet, по которой ведущие фирмы-производители программных средств в области коммуникаций интегрировали свои программные продукты с ОС VxWorks. На сегодняшний день — это сетевые протоколы X.25, ISDN, ATM, SS7, Frame Relay и OSI; CASE-средства разработки распределенных систем на базе стандартов ROOM (Real-Time Object Oriented Modelling) и CORBA (Common Object Request Broker Architecture); менеджмент сетей по технологиям MBD (Management By Delegation) и CMIP/GDMO (Common Management Information Protocol/Guidelines for Definition of Managed Objects).

ОС VxWorks применяется в самых различных областях, приведем их небольшой перечень:

− Phoenix Mars Lander — аппарат, предназначенный для изучения Марса.

− Зонды Spirit и Opportunity, а также аппарат Mars Reconnaissance Orbiter используют ОС VxWorks на платформе POWER. Система используется и в других космических миссиях, например Deep Impact.
− Планируется использование в новейших авиалайнерах Boeing 787 и Boeing 747-8.
− Коммуникационное оборудование многих компаний (например, Nortel, 3COM, Alcatel и др.).
− Linksys WRT54G (ver.5,6,…), NetGear WGR614 (ver. 5,6,7).
− Некоторые PostScript-принтеры.
− Медицинское оборудование компании Siemens AG (в частности, магнитно-резонансные томографы).
− Последние системы интерфейсов BMW iDrive.
− Система управления робототехническими комплексами компании KUKA.

На основе VxWorks разработана весьма популярная и достаточно мощная ОС **VSPWorks.** VSPWorks обеспечивает многозадачный режим с приоритетами и поддержку быстрых прерываний на процессорах DSP и ASIC. ОСРВ VSPWorks следует модели единственного виртуального процессора, что значительно упрощает распределение приложений в многопроцессорной системе, сохраняя при этом производительность жесткого реального времени. VSPWorks является модульной и масштабируемой.

ОСРВ VSPWorks обладает многослойной структурой (рис. 5.4), что служит хорошей основой для абстрагирования и переносимости. Центром системы служит сильно оптимизированное наноядро (nanokernel), которое способно управлять совокупностью процессов. Ниже наноядра находятся программы, обслуживающие прерывания, выше наноядра располагается микроядро, которое управляет многозадачным режимом с приоритетами C/C++ задач [7].

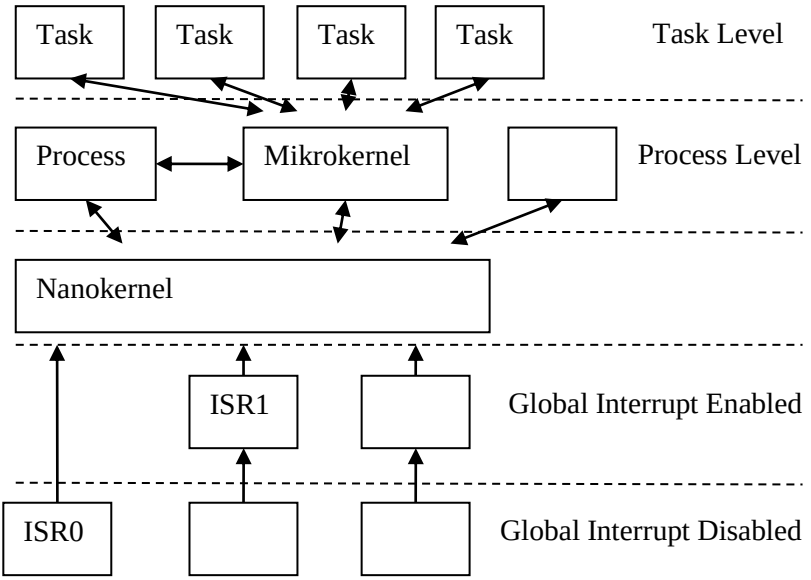

Рис. 5.4. Архитектура ОС РВ VSPWorks

Управление прерываниями имеет два уровня. Нижний уровень (уровень 0) используется для обработки аппаратных прерываний. Во время обработки таких прерываний все остальные прерывания блокируются. Код, выполняющийся на этом уровне, написан на языке ассемблера, и ответственность за сохранение соответствующих регистров в стеке ложится на программиста. На этом уровне может быть обработано прерывание, которое требует малого времени для обработки. Если обработка прерывания является более сложной и требует большего времени, то прерывание обрабатывается на более высоком уровне (уровень 1), где разрешено прерывание прерывания и, таким образом, они могут быть вложенными. Переход на более высокий уровень прерываний происходит по системному вызову [7].

Процессы наноядра (уровень 2) пишутся на языке ассемблера и имеют сокращенный контекст (т.е. используют меньше регистров). Эти процессы могут быть загружены и разгружены с процессора очень быстро. Каждому процессу присваивается

приоритет. Уровень 3 идеален для написания драйверов для интерфейсов аппаратуры низкого уровня [7].

Микроядро находится на уровне 3. Микроядро написано на языке С и имеет свыше 100 сервисов. Обработка задач на этом уровне ведется в режиме приоритетного прерывания, и планирование управляется приоритетами [7].

5.3.4. SoftKernel

Система SoftKernel выпускается фирмой Microprocess (France).

Основные характеристики:

Категория: host/target.

Архитектура: на основе объектов-микроядер.

Стандарт: собственный.

ОС разработки (host): Unix/Windows

Процессоры (target): Motorola 68xxx, Intel 80960, ARM, PowerPC, ARM, SPARC.

Линии связи host/target: последовательный канал и Ethernet.

Размер ядра: 40Кб.

Средства синхронизации и взаимодействия: семафоры, сигналы, события, почтовые ящики.

Планирование: приоритетное, preemptible.

Средства разработки:

SoftWorks — интегрированная среда разработки на C/C++.

ОС SoftKernel основана на идеи строить саму СРВ, используя объектно-ориентированный подход. При этом в ее основу заложены следующие положения:

− как приложение, так и операционная система полностью объектно-ориентированы и используют все преимущества этого подхода;

− приложение и ОСРВ могут быть полностью интегрированы, поскольку используют один объектно-ориентированный язык программирования;

− обеспечивается согласование интерфейсов ОСРВ и приложения;

− приложение может «моделировать» ОСРВ для своих потребностей, заказывая нужные ему объекты;
− единый комплекс (приложение + ОСРВ) является модульным и легко модернизируемым.
− ОС SoftKernel, целиком написана на C++.

По принципу своей работы инструментальные средства Soft Kernel и Soft Environment напоминают пакет, служащий буфером между операционной системой и тем или иным аппаратным модулем (BSP — board support package), только в данном случае в роли такого модуля выступает интерфейс 32-разрядных Windows-приложений (Win32 API). Запустив одновременно несколько инструментов Soft Kernel на одной хост-машине, можно создать модель сетевой или многопроцессорной среды и вести разработку соответствующего проекта.

OSE Soft Kernel позволяет OSE real-time процессам быть запущенными на компьютере. При помощи неё ускоряется разработка и тестирования приложений без предварительного переноса приложения на железо. Это симулятор однопроцессорной системы (single CPU system). Приложения могут быть расположены на машине, так как будто они оба запущены на OSE системе.

Традиционно имеется жёсткая граница между симуляцией и операцией в СРВ.

Графическое представление OSE Soft Kernel интегрированно с OSE Illuminator, инструментом разработки и отладки. Illuminator имеет дружеский графический интерфейс, который предоставляет лёгкий доступ к его инструментам и плагинам.

Дополнительно OSE Soft Kernel позволяет пользователям отлаживать приложения в debug-режиме непосредственно сгенерировав код. Это намного эффективней, как если бы отладка производилась на железе. Необходимо было бы пользователю загрузить, зафиксировать ошибку, поправить ошибку, загрузить снова и проверить, нет ли каких либо ошибок ещё.

Приложения, запущенные под OSE Soft Kernel, могут использовать все системы совместимые с real-time kernel.

5.3.5. CHORUS

Система CHORUS выпускается фирмой Chorus Système (France). В ноябре 1997г. фирма приобретена компанией Sun Microsystems (USA).

Основные характеристики:

Категория: host/target (CHORUS/Micro и CHORUS/ClassiX) и self-hosted (CHORUS/MiX).

Архитектура: на основе микроядра.

Стандарт: собственный и POSIX 1003.

ОС разработки (host): CHORUS/Unix/Windows

Процессоры (target): Intel 80x86, Motorola 68xxx, Motorola 88xxx, SPARC, PowerPC, T805, MIPS, PA-RISC, YMP (Cray), DEC Alpha.

Линии связи host/target: последовательный канал и Ethernet.

Размер ядра: 10Кб для CHORUS/Micro, 50Кб для CHORUS/ClassiX.

Средства синхронизации и взаимодействия: POSIX 1003.

Планирование: приоритетное, FIFO, preemptible (вытесняющая).

Средства разработки:

CHORUS/Harmony — интегрированная среда разработки C/C++, включающая компиляторы, отладчик, анализатор.

CHORUS/JaZZ — виртуальная машина Java.

ОС Chorus — это масштабируемая встраиваемая ОС, широко применяемая в телекоммуникационной индустрии.

ОС Chorus поддерживает на одной аппаратной платформе широкий набор телекоммуникационных протоколов, унаследованных приложений, приложений режима реального времени и Java-технологии.

ОС Chorus моделирует три сорта приложений:

– ***POSIX-процессы*** составляют большинство приложений Chorus; эти приложения имеют доступ к чисто POSIX API, нескольким POSIX-подобным расширенным API и небольшому числу ограниченных системных вызовов микроядра;

– ***Акторы Chorus*** — эти приложения выполняются над микроядром и ограничиваются API микроядра, акторы включают драйверы, события подсистем и протокольные стеки;
– ***Унаследованные приложения Chorus*** поддерживаются для совместимости с приложениями, разработанными для более ранних версий Chorus.

Архитектура ОС Chorus является многослойной, основанной на компонентах. Микроядро содержит минимальный набор компонентов, необходимых для функционирования ОС:
− компонент kern реализует интерфейс микроядра и содержит актор KERN, вспомогательную библиотеку и заголовочные файлы;
− компонент pd — менеджер приватных данных реализует интерфейс между подсистемами микроядра;
− компонент pmm — менеджер постоянной памяти реализует интерфейс постоянной памяти;
− компонент core executive обеспечивает существенную часть поддержки реального времени.

Компонент диспетчера ядра (core executive) обеспечивает следующую функциональность:
− поддержка многочисленных независимых приложений;
− поддержка пользовательских и системных приложений;
− поддержка актора — единицы модуляризации приложений;
− поддержка единицы исполнения — потока;
− операции управления потоками;
− управление точкой локального доступа (LAP — Local Access Point);
− сервисы управления исключительными ситуациями;
− минимальный сервис управления прерываниями.

В core executive отсутствует управление такими сущностями, как синхронизация, планирование, время, память. Политики управления этими понятиями обеспечиваются дополнительными компонентами, которые выбираются пользователем в зависимости от требований аппаратных и программных средств. Core executive всегда присутствует в исполняемом экземпляре ОС Chorus, остальные компоненты конфигурируются и добавляются по необходимости. Размер резидентной часть ядра составляет 10Kb.

Понятие «актор» в ОС Chorus определяется как единица загрузки для приложения. Оно также служит единицей инкапсуляции для того, чтобы сопоставить все системные ресурсы, используемые приложением, и потоки, выполняющиеся внутри актора. Примерами таких ресурсов являются *потоки, регионы памяти и конечные точки взаимодействия*.

Микроядро Chorus во многих отношениях походит на реализации Mach, выполненные IBM и OSF. ОС Chorus включает поддержку распределенных процессоров, нескольких распределенных серверов операционной системы (во многом похожую на комбинацию Mach-OSF/1), управления памятью и обработки прерываний. Поддерживается также прозрачное взаимодействие с другими экземплярами микроядра ОС Chorus, что делает Chorus хорошей основой для сильно распределенных систем.

Существует несколько реализаций микроядра ОС Chorus. Chorus/MiX, версия компании Chorus операционной системы с интерфейсами UNIX, включает отдельные версии, совместимые с SVR3.2 и SVR4. Специально для использования на персональных компьютерах компания Chorus поддерживает реализацию Chorus/MiX, совместимую с SCO.

ОС Chorus работает на самых разных процессорах от Intel 80x86 до транспьютеров Inmos. Motorola объявила о разработке спецпроцессора семейства PowerPC, ориентированного на ядро Chorus и предназначенного для применения во встроенных системах.

Операционные системы компании Chorus основаны на минимальном ядре (50-60 КБайт), в функции которого входят планирование, управление памятью, обработка событий реального времени и управление коммуникациями. Остальные функции операционной системы выполняются серверами, которые находятся вне ядра, взаимодействуя с ним путем передачи сообщений. Файловые менеджеры, менеджеры потоков и сокетов и даже драйверы устройств организуются в виде серверов. Группа таких серверов называется подсистемой.

Драйверы устройств не включаются в ядро. Аналогично подходу IBM, драйверы получают доступ к аппаратуре через ядро. Это дает возможность компоненту более высокого уровня

(менеджеру устройств) отслеживать работу драйверов, функционирующих в разных узлах распределенной системы.

Ядро ОС Chorus состоит из нескольких функциональных частей. Многозадачная исполнительная система распределяет локальные процессоры и планирует выполнение нитей на основе системы приоритетов. Программный интерфейс исполняющей системы предоставляет примитивы создания и уничтожения нитей, синхронизации с использованием семафоров, блокировок, условных переменных и т.д.

Менеджер памяти поддерживает механизм распределенной виртуальной памяти. Базовой единицей хранимых данных является сегмент, который обычно располагается на каком-либо устройстве внешней памяти. Виртуальное адресное пространство актора (аналога процесса в терминах ОС Chorus) разбивается на непрерывные участки, каждый из которых служит для отображения части сегмента в физическую память. Специальные системные акторы управляют сегментами, поддерживая согласованность распределенной разделяемой памяти при параллельном доступе к сегменту нескольких нитей. Супервизор заведует начальной обработкой прерываний и их переадресовкой динамически загружаемым драйверам устройств и другим обработчикам событий.

Менеджер межпроцессных взаимодействий передает сообщения между акторами в пределах одного узла; отделенный от ядра сетевой менеджер обслуживает связь с удаленными портами на основе сетевых коммуникаций. Кроме того, сетевой менеджер оказывает прямые транспортные услуги тем акторам, которым они требуются. Для обычных акторов сетевая природа взаимодействия процессов прозрачна.

Единственные аппаратно-зависимые компоненты ядра — супервизор и часть менеджера памяти, что делает ядро ОС Chorus очень мобильным.

Проект COOL (Chorus Object-Oriented Layer) направлен на развитие объектно-ориентированных средств. Определены три слоя над ядром ОС Chorus.

Первый слой, COOL-база, инкапсулирует ядро ОС Chorus, образуя новое объектно-ориентированное микроядро с интерфейсом системных вызовов. COOL-база имеет дело с кластера-

ми — абстракциями наборов областей виртуальной памяти. С точки зрения компонентов COOL более высокого уровня, кластер — это область существования объекта. COOL-база управляет кластерами, отображая их в многочисленные адресные пространства и создавая тем самым распределенное пространство кластеров.

Над слоем COOL-базы находится слой GRT (generic runtime), обеспечивающий выполнение объектов, виртуальную память объектов, одноуровневое хранение постоянных объектов, RPC-взаимодействия (Remote Procedure Call) объектов и подсистему защиты объектов.

На самом верхнем слое COOL обеспечивается поддержка для конкретных языков.

Взаимодействие процессов на основе передачи сообщений — это простой и элегантный способ организации распределенных систем. Однако принято считать, что он менее эффективен, чем разделяемая память. В любой системе, в которой компоненты взаимодействуют только посредством сообщений, потери при их передаче серьезно влияют на общую производительность системы.

Сообщения. В данном случае сообщения — это нетипизированные строки; менеджер IPC (Inter-Process Communication) не контролирует поток сообщений и не проверяет их правомерность. Эти средства добавляются на уровне подсистем, так что дополнительные накладны расходы возникают только в тех случаях, когда это необходимо.

В реализации механизма RPC учитывается возможность локального взаимного расположения клиента и сервера. Если, например, клиент и сервер находятся в одном узле, менеджер IPC при посредстве менеджера памяти передает сообщение с помощью простого изменения отображения адресов виртуальной памяти, без реального копирования данных.

В ранней версии ОС Chorus интерфейс Unix полностью основывался на передаче сообщений; требовалось также, чтобы все драйверы устройств являлись частью ядра и выполнялись в привилегированном режиме. Поэтому все процессы подсистемы Unix содержали переходники для преобразования системных вызовов в сообщения, что препятствовало двоичной совмести-

мости с Unix System V. В настоящее время введены акторы супервизора, выполняемые в пространстве супервизора в привилегированном режиме, но компилируемые и загружаемые как отдельные модули. Только им предоставляется прямой доступ к аппаратным средствам; такие подключаемые обработчики могут образовывать нити, вызываемые из ядра как подпрограммы и возвращающие управление ядру. Их наличие позволяет обеспечить традиционный интерфейс с ядром на основе внутренних прерываний (вместо передачи сообщений) и добиться реальной двоичной совместимости с System V. Одновременно резко сокращается время реакции на прерывание и появляется возможность реализовывать драйверы устройств полностью вне ядра.

С 2002 г. компания Chorus Systems открыла коды Chorus и перестала поддерживать систему. Участники группы Chorus из Sun Microsystems (среди которых нет участников оригинального проекта Chorus) основали компанию VirtualLogix, занимающуюся средствами виртуализации систем реального времени (возможно, на основе Chorus).

5.3.6. pSOS

Система pSOS выпускается фирмой Integrated Systems (USA). В феврале 2000г. фирма приобретена компанией Wind River Systems (USA).

Основные характеристики:

Категория: host/target.

Архитектура: на основе микроядра.

Стандарт: собственный.

ОС разработки (host): Unix/Windows

Процессоры (target): Intel 80x86, Motorola 68xxx, Intel 80960, ARM, PowerPC, MIPS.

Линии связи host/target: последовательный канал и Ethernet.

Размер ядра: 15Кб.

Средства синхронизации и взаимодействия: семафоры, события, mutex и д.р.

Планирование: приоритетное, preemptible.

Средства разработки:

Интегрированная среда разработки C/C++/Ada.

Имя **pSOSsystem** присвоено операционной системе, имя **pSOS+** — ее ядру. **pRISM+** — это интегрированная среда разработки для создания приложений.

pSOS+ — это маленькое ядро встраиваемых приложений, представляющее собой некий вариант клиент-серверной архитектуры. Однако оно не имеет протокола взаимодействия, основанного на сообщениях. Для взаимодействия модулей используется программная шина (software bus). Есть возможность выбрать и встроить модули в систему во время компиляции. Такими модулями могут быть файловая система (pHILE+), отладчик (pROBE+), сетевые протоколы (pNA+), библиотека удаленных вызовов процедур (pRPC+) и стандартная библиотека ANSI C (pREPC+).

Вызовы различных приложений осуществляются через программные прерывания.

pSOS+m является многопроцессорной версией ядра **pSOS+**. Она требует, чтобы один узел был главным, а остальные подчиненными. К этому ядру добавлены системные вызовы, позволяющие оперировать через границы процессора.

В **pSOS+** не используется понятие процесса, вместо этого она оперирует задачами, что соответствует понятию потоков, выполняющихся в одном процессе. Все системные объекты разделяются между всеми потоками. Так как все потоки разделяют один и тот же контекст, время переключения потоков становится очень малым.

pSOSsystem имеет несегментированную модель памяти. Защита памяти может быть обеспечена через библиотеку управления памятью. Код, данные и стеки можно защитить с помощью определения отображений защиты памяти для каждой задачи. При этом ответственность ложится на разработчика приложений, а это является непростой задачей. pSOSsystem предлагает две абстракции для управления памятью: регионы и разделы. Регионы — это куски памяти нефиксированного размера, в то время как разделы — куски фиксированного размера. Управление памятью с помощью разделов обеспечивает быстрое выделение памяти.

Управление прерываниями в pSOSsystem довольно примитивное. Кроме того, отсутствуют мьютексы и механизм наследования приоритетов, что может привести к инверсии приоритетов.

5.3.7. ОС2000

Система ОС2000 заработана научно-исследовательским институтом системных исследований при Российской Академии Наук (НИИСИ РАН).

Основные характеристики:

Категория: host/target.

Архитектура: на основе микроядра.

Стандарт: POSIX 1003.

ОС разработки (host): UNIX (Linux).

Процессоры (target): Intel 80x86, MIPS.

Линии связи host/target: VNE, Ethernet (протоколы семейства TCP/IP).

Средства синхронизации и взаимодействия: POSIX 1003.

Средства разработки: Языки С, Ассемблер.

ОС2000 (ОС РВ «Багет 2.0» — микроядерная исполнительская операционная система реального времени (ОС РВ) для процессоров Intel и MIPS. Система сертифицирована министерством обороны РФ для использования в военной вычислительной технике. Поставляется вместе с компьютерами серии Багет. ОС 2000 считается первой серьезной отечественной операционной системой для встроенных систем. Интерфейс и дизайн системы похож на ОС РВ VxWorks компании Wind River System. Есть мнение, что реализация ОС2000 является полностью российской, существует только совместимость по системным вызовам.

Операционная система ОС2000 была разработана в 1998-2001 гг. К настоящему моменту выпущено три издания ОС2000. В качестве основы для пользовательского интерфейса был выбран стандарт POSIX 1003.1 (X Window). Взаимодействие по сети (аппарат сокетов системы FreeBSD) также реализован в соответствии со стандартом POSIX 1003.1.

В качестве основного языка программирования ОС2000 предполагается использовать язык С.

Разработка операционной системы реального времени ОС2000 базируется на следующих основополагающих принципах:

- соответствие международным стандартам;
- мобильность;
- использование концепции микроядра;
- использование объектно-ориентированного подхода;
- использование свободно распространяемого программного обеспечения;
- распространение ОС вместе со средствами разработки прикладных программ.

Для систем реального времени типичным является совместное использование ресурсов, поэтому в ОС2000 реализованы только потоки управления, но не процессы (с точки зрения POSIX в системе имеется только один процесс).

Для эмуляции протокола Ethernet в многопроцессорных системах, использующих шину VME, использован стандарт ANSI/VITA. Этот стандарт позволяет взаимодействовать через шину VME и общую память различным как по аппаратуре, так и по используемой операционной системе процессорным платам.

В настоящее время ОС2000 содержит пакет поддержки модуля для ЭВМ серии «Багет» с процессором 1В578 (совместимого с процессором MIPS R3000) и для PC-совместимых компьютеров (с процессором Intel).

Архитектура программного обеспечения

При использовании ОС2000 программное обеспечение системы реального времени состоит из операционной системы и прикладной программы. В системах реального времени граница между операционной системой и прикладной программой не так резко очерчена, как в случае традиционных операционных систем. В частности, в системах реального времени прикладная программа может непосредственно вызывать те функции, которые в случае традиционных ОС могут выполняться только операционной системой (например, запрет или разрешение прерываний). Такая «свобода» позволяет повысить эффективность си-

стемы реального времени, но накладывает большую ответственность на прикладного программиста.

Для процессоров с архитектурой MIPS прикладная программа выполняется в режиме ядра и виртуальная память не применяется. Это позволяет заметно сократить время обращения к функциям операционной системы, сократить время переключения контекста и увеличить скорость выполнения прикладных программ и сделать ее предсказуемой.

Прикладная программа представляет собой совокупность потоков управления (пользовательских потоков управления).

При инициализации системы порождается корневой поток управления прикладной программы, при необходимости другие потоки управления прикладная программа порождает динамически.

Операционная система состоит из ядра и системных потоков управления. Ядро выполняет функции планирования, синхронизации и взаимодействия потоков управления, а также низкоуровневые операции ввода/вывода. Функции ядра выполняются в контексте вызвавшего его потока управления или функции обработки прерывания. Микроядро представляет собой небольшую часть ядра ОС, функциями которой пользуются другие части ОС. Микроядро содержит функции управления потоками нижнего уровня (включая планировщик) и быстрые средства синхронизации. Все другие функции (например, захват и освобождение семафора, низкоуровневые операции ввода/вывода) выполняются вне микроядра, используя его функции.

Системные потоки выполняют более сложные функции операционной системы, такие как ввод/вывод информации по сети или обмен информацией с файловой системой. Использование системных потоков для сложных и протяженных во времени функций ОС позволяет продолжать работу параллельно с выполнением этих функций. В рамках таких потоков можно выполнять часть функций, которые обычно выполняются в рамках обработчиков прерываний драйвера. *Во-первых*, такой подход дает определенные удобства при программировании, так как не все функции ОС, доступные в контексте потока, можно выполнять в контексте прерывания. *Во-вторых*, сокращается время

выполнения функций обработки прерываний, что особенно важно для систем реального времени. Действительно, если низкоприоритетная задача ведет интенсивный ввод/вывод, то она будет мешать выполнению высокоприоритетной задачи, так как функция обработки прерывания драйвера более приоритетна, чем любой поток управления. Потери (времени) будут тем больше, чем больше времени будет занимать функция обработки прерываний.

Для разработки прикладного программного обеспечения используется комплекс, состоящий из двух ЭВМ, соединенных по сети:

- инструментальная ЭВМ (компьютер с операционной системой типа UNIX (Linux));
- целевая ЭВМ (ЭВМ, для которой разрабатывается программное обеспечение).

Разработка программного обеспечения ведется на инструментальной ЭВМ.

Средства разработки позволяют оттранслировать программу, написанную на языках С и Ассемблер, а также отлаживать программу, загруженную в целевую машину.

5.3.8. QNX

Система QNX выпускается фирмой QNX SoftWare Systems (Canada & USA).

Основные характеристики:

Категория: self-hosted.

Архитектура: на основе микроядра.

Стандарт: POSIX 1003.

Процессоры (target): Intel 80x86.

Линии связи host/target: последовательный канал и Ethernet.

Размер ядра: 60Кб.

Средства синхронизации и взаимодействия: POSIX 1003.

Планирование: приоритетное, FIFO, адаптивное (спорадическое), Roun Robin, preemptable.

Средства разработки:

Комплект разработки C/C++, включающая компиляторы, отладчик, анализатор от QNX и независимых поставщиков (Watcom, SyBase).

X Windows/Motif для QNX.

Комплект разработки QNX Momentics.

Операционная система QNX является мощной операционной системой, позволяющей проектировать сложные программные системы, работающие в реальном времени, как на одном-единственном компьютере, так и в локальной вычислительной сети. Встроенные средства операционной системы QNX обеспечивают поддержку многозадачного режима на одном компьютере и взаимодействие параллельно выполняемых задач на разных компьютерах, работающих в среде локальной вычислительной сети. Основным языком программирования в системе является язык С. Основная операционная среда соответствует стандартам POSIX-интерфейса. Это позволяет с небольшими доработками перенести необходимое накопленное программное обеспечение в среду операционной системы QNX для организации их работы в среде распределенной обработки.

ОС QNX является сетевой, мультизадачной, многопользовательской (многотерминальной) и масштабируемой. С точки зрения пользовательского интерфейса и API она очень похожа на UNIX. Однако, QNX – это не версия UNIX, хотя почему-то многие так считают. Она была разработана, что называется «с нуля», канадской фирмой QNX Software Systems Limited в 1989 году по заказу Министерства обороны США [8]. Причем эта система построена на совершенно других архитектурных принципах, отличных от принципов, использованных при создании ОС UNIX.

QNX была первой коммерческой ОС, построенной на принципах микроядра и обмена сообщениями. Система реализована в виде совокупности независимых (но взаимодействующих через обмен сообщениями) процессов различного уровня (менеджеры и драйверы), каждый из которых реализует определенный вид сервиса. Эти идеи позволили добиться нескольких важнейших преимуществ. Вот как они описаны на сайте, посвященном системе QNX.

- *Предсказуемость,* означающая ее применимость к задачам жесткого реального времени. QNX является операционной системой, которая дает полную гарантию в том, что процесс с наивысшим приоритетом начнет выполняться практически немедленно и что критическое событие (например, сигнал тревоги) никогда не будет потеряно. Ни одна версия UNIX не может достичь подобного качества, поскольку нереентерабельный код ядра слишком велик. Любой системный вызов из обработчика прерывания в UNIX может привести к непредсказуемой задержке (то же самое касается Windows NT).
- *Масштабируемость* и *эффективность,* достигаемые оптимальным использованием ресурсов и означающие ее применимость для встроенных (embedded) систем; вы не увидите в каталоге /dev огромной кучи файлов, соответствующих ненужным драйверам. Драйверы и менеджеры можно запускать и удалять (кроме файловой системы, что очевидно) динамически, просто из командной строки. Вы можете иметь только тот сервис, который вам реально нужен, причем это не требует серьезных усилий и не порождает проблем.
- *Расширяемость* и *надежность* одновременно, поскольку написанный вами драйвер не нужно компилировать в ядро, рискуя вызвать нестабильность системы. Менеджеры ресурсов (сервис логического уровня) работают в кольце защиты 3, и вы можете добавлять свои, не опасаясь за систему. Драйверы работают в кольце с уровнем привилегий 1 и могут вызвать проблемы, но не фатального характера. Кроме того, их достаточно просто писать и отлаживать.
- *Быстрый сетевой протокол FLEET,* прозрачный для обмена сообщениями, автоматически обеспечивающий отказоустойчивость, балансирование нагрузки и маршрутизацию между альтернативными путями доступа.
- *Компактная графическая подсистема Photon,* построенная на тех же принципах модульности, что и сама ОС, позволяет получить полнофункциональный GUI (расширенный Motif), работающий вместе с POSIX-совместимой ОС всего в 4Мбайт памяти, начиная с i80386 процессора.

В связи с тем, что ОСРВ QNX является одной из самых распространенных операционных систем реального времени на территории нашего государства, в последующих главах учебного пособия ее строение и функциональные возможности будут рассмотрены более подробно. В качестве излагаемой базовой версии будем использовать ОС РВ QNX 6.3 [9].

5.4. Специализированные ОСРВ

Системы, проектируемые под конкретную модель микроконтроллера или конкретную задачу, обладает определенными преимуществами:

– наивысшая производительность;
– наилучший учет особенностей оборудования;
– наибольшая компактность.

Недостатки специализированных ОСРВ:

– большое время разработки;
– высокая стоимость;
– непереносимость на другие платформы.

Примерами таких систем являются ОСРВ, разработанные многими производителями электронной техники (Sony, Sagem, …), а также системы, разработанные под конкретную большую задачу, например, управление железными дорогами (TGV, France).

5.5. Обобщенный обзор функциональности ОСРВ

Количество операционных систем реального времени в настоящий момент на мировом и российском рынках весьма велико. Описанный выше ряд операционных систем можно продолжить весьма большим перечнем: Parallel C (3L Limited), Nucleus Plus/RTX (Accelerated Technology), RISC OS (Acorn Risc Technologies), Allegro (Allegro Systems Limited), ARTK (Alsys GmBH), ARTA (Apple), Rubus OS (Arcticus Systems), VCOS (AT&T), XTAL RTOS (AXE), AmpOS (Belobox Systems), BOX (Brainstorm Engineering), Byte_Bos Multitasking (Byte Bos Inte-

grated Systems), MIPS Kernel (CAC), UNOS (Charles River Data Systems), CMX (CMX Company), «PowerMAX, MAXION» (Concurrent Computer), Digital UNIX (Digital Equipment) и т.д. И это малая часть всего перечня.

Подведем итог описания ОСРВ сводной таблицей показывающих наименование фирму производителя и их основные функциональные возможности (таблица 5.3). В таблице показаны как ОСРВ вошедшие в выше приведенный обзор, так и не вошедшие.

Существующие ОСРВ обслуживают довольно внушительный парк микропроцессоров. Приведем некоторые наименования микропроцессоров из этого парка: Intel Pentium, Intel 80x86, NEV V53, NEV V25, Z 80 (180), 80x96, 80x51, i960, i860, 68060, 68040, 68030, 68020, 68010, 68000, Z180, 68340, 68332, 68331, 68302, PowerPC6604, PowerPC6603, PowerPC6601, Power PC 403, 68HC16, 68HC11, MIPS R4000, 68360, AM294xx, AM292xx, AM290xx, SPARC, VA X family, Alpha, Inmos T4xx, T8xx, ARM, ARM THUMB, TMS320C5x, TMS320C4x, TMS320C3x, TMS320C2x, 56xxx DSP, AD2106x и др.

Таблица 5.3. Обзор функциональных возможностей ОСРВ

Название ОС (Компания)	**Исходные тексты**	**Поддержка WinAPI**	**Разработка под Windows**	**Поддержка POSIX API**	**Работа из ПЗУ**	**Фиксированные приоритеты**	**Приоритетная многозадачность**	**Защита памяти**	**Модульность**	**Многопроцессорность**	**Сетевые средства**	**Межзадачные коммуникации**	**Связь через общую память**	**Межзадачная синхронизация**	**Удаленный вызов процедур**	**Удаленный запуск задач**	**Удаленный мониторинг**	**Удаленная отладка**
RTX (VenturCom)		X	X	X	X	X	X	X	X	X	X	X	X	X	X			
ETS Kernel (Phar Lap Software)	X	X	X		X		X	X			X	X		X			X	X
Hyperkernel (Nematron)		X	X															
LynxOS	X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

(LynuxWorks)																		
OS 9 (Microware Systems)	X		X	X	X		X	X	X	X	X	X	X	X	X	X	X	X
«VxWorks, Tornado» (Wind River Systems)	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
SoftKernel (Microdata Soft)		X	X		X		X	X	X	X	X	X		X			X	X
CHORUS/ClassiX (Chorus Systems)			X	X		X		X	X		X	X	X	X	X	X	X	X
«pSOS+, FlexOS» (Integrated Systems)	X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
QNX (QNX Software Systems)	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X
Nucleus Plus/RTX (Accelerated Technology)	X		X		X		X		X		X	X		X				
Allegro	X						X	X	X		X	X	X					

(Allegro Systems Limited)																		
ARTK (Alsys GmBH)	X			X	X	X	X	X	X	X	X	X	X		X			X
VCOS (AT&T)						X			X	X		X	X	X				
BOX (Brainstorm Engineering)			X		X	X	X	X	X	X	X	X	X	X	X	X	X	X
MIPS Kernel (CAC)	X						X	X	X	X		X	X	X		X		X
UNOS (Charles River Data Systems)	X			X	X	X	X	X		X	X	X	X	X	X	X	X	X
CMX (CMX Company)	X	X	X		X		X	X	X		X	X	X	X			X	X
RTXC Embedded Systems Products)	X		X		X	X	X	X	X		X	X	X	X	X	X	X	X
HP RT (Hewlett Packard)	X			X			X	X	X	X	X		X		X			X
OS/Open (IBM)			X	X	X	X			X		X	X		X	X			X
VRTX (Microtec Research)			X		X		X			X	X	X		X	X			
RMOS (Siemens)		X	X		X		X	X	X	X	X	X	X	X				X

Вопросы для самопроверки

1. На каких процессорах может работать ОС Linux?
2. Какие способы существуют для реализации механизма preemption?
3. В чем особенность системы RT-Linux?
4. Приведите основные аргумента «За» и «Против» использования Windows NT в качестве ОСРВ.
5. Какие расширения Windows NT, которые поддерживают реальное время, вы знаете?
6. Классифицируйте в виде таблицы собственно ОСРВ по категориям: self-hosted, host/target и те которые могут выступать и в том и другом варианте.
7. Опишите основные функциональные возможности ОС LynxOS.
8. Какие механизмы синхронизации реализованы в ОС LynxOS 5.0?
9. Опишите реализацию атомарных переменных и операций в ОС LynxOS 5.0.
10. Какие барьеры памяти реализованы в ОС LynxOS 5.0?
11. Опишите ОС функциональные возможности OS-9.
12. Опишите ОС функциональные возможности VxWorks.
13. Опишите уровневую структуру VSPWorks.
14. Опишите ОС функциональные возможности SoftKernel.
15. Опишите ОС функциональные возможности CHORUS.
16. Опишите ОС функциональные возможности pSOS.
17. Опишите ОС функциональные возможности ОС2000.
18. Приведите преимущества и недостатки специализированных ОСРВ.

6. Микроядро ОС QNX Neutrino

6.1. Введение

Микроядро **ОС QNX Neutrino** реализует основные функции стандартов POSIX, используемые во встраиваемых системах реального времени, а также базовые службы обмена сообщениями. Те функции, которые не реализованы в микроядре, выполняются дополнительными процессами или разделяемыми библиотеками.

На самых нижних уровнях микроядра расположено всего несколько базовых объектов, а также четко отрегулированные процедуры для управления ими. Это является основой, на которой строится ОС QNX (рис. 6.1).

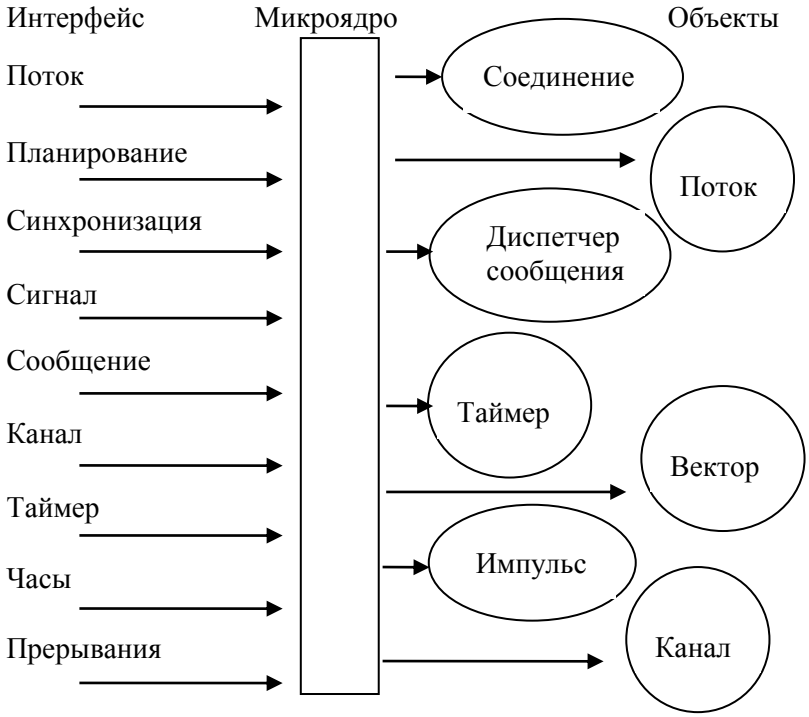

Рис. 6.1. Микроядро ОС QNX Neutrino

ОС QNX Neutrino большинство служб по обеспечению работы в реальном времени и по планированию потоков реализовано прямо в микроядре. Эти службы могут работать без использования дополнительных модулей.

В микроядре ОС QNX существуют системные вызовы (kernel calls), которые служат для управления следующими объектами:

- потоками;
- сообщениями;
- сигналами;
- часами;
- таймерами;
- обработчиками прерываний;
- семафорами;
- блокировками взаимного исключения, мютаксами;
- условными переменными;
- барьерами.

ОС QNX целиком построена на основе таких вызовов, причем ядро ОС полностью вытесняемо (preemtable), в том числе и во время обмена сообщениями между процессами.

В микроядро включены службы, которые порождали наиболее короткую ветвь исполняемого кода. Операции, требующие значительных ресурсов (например, загрузка процесса), были переданы внешним процессам и потокам, в которых работа по переключению на контекст другого процесса пренебрежительно мала в сравнении с работой по обработке запроса, выполняемой внутри этого потока.

Оценим работу, которая выполняется между переключениями контекстов во время обмена сообщениями, и учтем высокую скорость их переключений в минимизированном ядре. Очевидно, что объем времени, расходуемый на выполнение этих действий, настолько минимален, что становиться пренебрежительно малым в сравнении с работой, производимой для обработки запросов от механизма обмена межзадачными сообщениями.

На рис. 6.2. показан механизм вытеснения в ядре для процессоров с архитектурой Intel x86 без поддержки SMP. Запрет прерываний или отсутствие вытеснения происходит только на короткие периоды времени (как правило, эти периоды не превышают порядка сотен наносекунд).

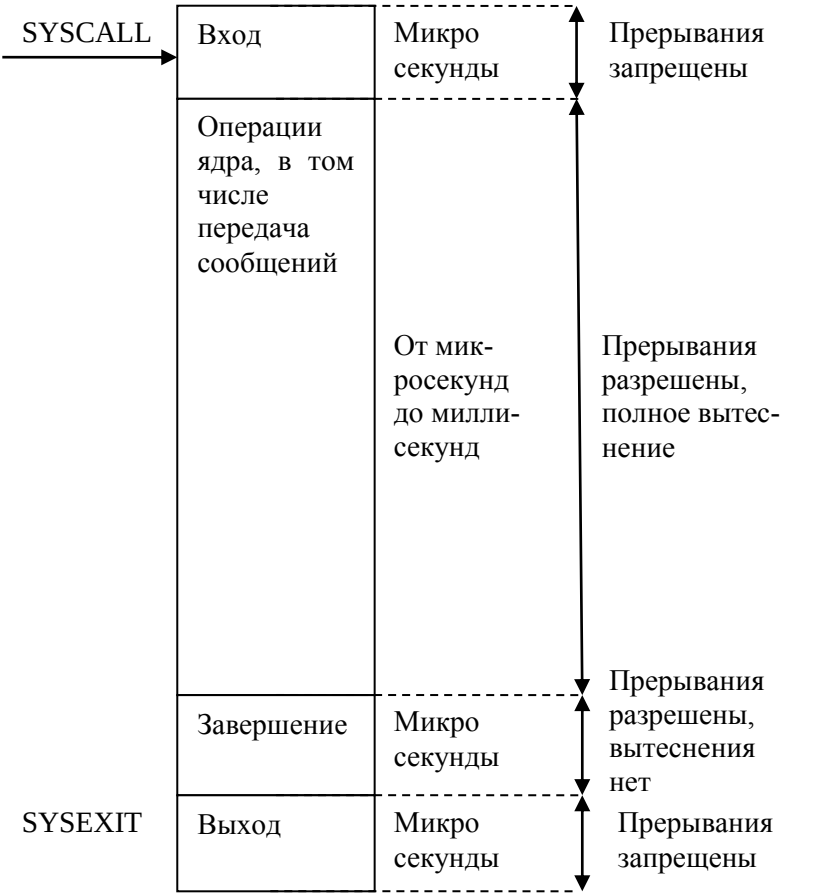

Рис. 6.2. Механизм вытеснения в ОС QNX

6.2. Потоки и процессы

6.2.1. Вызовы по управлению потоками

В ядре ОС QNX минимальной планируемой и выполняемой единицей является поток. Процесс можно рассматривать как объект, который содержит в себе эти потоки и который определяет для их выполнения свое адресное пространство.

В микроядре ОС QNX реализован ряд вызовов по управлению потоками POSIX и соответствующие им вызовы микроядра (таблица 6.1).

Таблица 6.1. Вызовы по управлению потоками POSIX и соответствующие вызовы микроядра [9]

POSIX-вызов	Вызов микроядра	Описание
pthread_create()	*ThreadCreate()*	Создать поток
pthread_exit()	*ThreadDestroy()*	Уничтожить поток
pthread_detach()	*ThreadDetach()*	Отсоединить поток, чтобы не ждать его завершения
pthread_join()	*ThreadJoin()*	Присоединить поток и ждать его кода завершения
pthread_cancel()	*ThreadCancel()*	Завершить поток в следующей точке завершения
отсутствует	*ThreadCtl()*	Изменить характеристики потока
pthread_mutex_init()	*SyncTypeCreate()*	Создать мьютекс
pthread_mutex_destroy()	*SyncDestroy()*	Уничтожить мьютекс
pthread_mutex_lock()	*SyncMutexLock()*	Блокировать мьютекс

pthread_mutex_trylock()	*SyncMutexLock()*	Условно блокировать мьютекс
pthread_mutex_unlock()	*SyncMutexUnLock()*	Снять блокировку мьютекса
pthread_cond_init()	*SyncTypeCreate()*	Создать условную переменную
pthread_cond_destroy()	*SyncDestroy()*	Уничтожить условную переменную
pthread_cond_wait()	*SyncCondvarWait()*	Ожидать условную переменную
pthread_cond_signal()	*SyncCondvarSignal()*	Разблокировать один из потоков, блокированных на условной переменной
pthread_cond_broadcast()	*SyncCondvarSignal()*	Разблокировать все потоки, блокированные на условной переменной
pthread_getschedparam()	*SchedGet()*	Получить параметры планирования и дисциплину потока
pthread_setschedparam()	*SchedSet()*	Установит параметры планирования и дисциплину потока
pthread_sigmask()	*SignalProcMask()*	Проверить и

		вывести маску сигналов потока
pthread_kill()	*SignalKill()*	Отправить сигнал потоку
sem_post()	*SyncSemPost()*	Увеличить значение счетчика на семафоре
sem_wait()	*SynSemWait()*	Уменьшить значение счетчика на семафоре

pthread_barrierattr_getpshared()	Получить значение атрибута совместного использования для заданного барьера
pthread_barrierattr_ destroy()	Уничтожить атрибутную запись барьера
pthread_barrierattr_init()	Инициализировать атрибуты объекта
pthread_barrierattr_ setpshared()	Установить значение атрибута совместного использования для заданного барьера
pthread_barrier_destroy()	Уничтожить барьер
pthread_ barrier_init()	Инициализировать барьер
pthread_barrier_wait()	Синхронизировать потоки на барьере
pthread_sleepon_lock()	Блокировать поток с использованием ждущей блокировки
pthread_rwlock_rdlock()	Блокировка потока по чтению/записи
pthread_rwlock_wrlock()	Блокировка потока по записи
pthread_rwlock_unlock()	Снятие блокировки по чтению
pthread_rwlock_tryrdlock()	Тестирование блокировки по чтению/записи
pthread_rwlock_trywrlock()	Тестирование блокировки по

	записи

Несмотря на то, что потоки внутри процесса используют одно адресное пространство, каждый из этих потоков имеет «собственные» данные. В некоторых случаях эти данные защищаются внутри ядра (например, идентификатор потока), в то время как другие данные остаются незащищенными в адресном пространстве процесса (например, у каждого потока свой стек).

Данные, относящиеся к потоку, реализуются в библиотеки ***pthread*** и хранятся в локальной памяти потока. Они обеспечивают механизм, предназначенный для связывания ***глобального целочисленного ключа процесса (process global integer key)*** с уникальным значением данных для каждого потока.

6.2.2. Состояния потока

Потоки создаются и уничтожаются динамически, и их количество внутри процесса может изменяться в значительных пределах. Создание потока включает в себя выделение и инициализацию необходимых ресурсов внутри адресного пространства процесса, а также запуск выполнения потока с некоторой функции.

Завершение потока включает в себя остановку потока и освобождение ресурсов потока.

Поток ОС QNX может находиться в одном из следующих состояний [9]:

− CONDVAR — поток блокирован на условной переменной (например, при вызове функции pthread_cond_var_wait());

− DEAD — поток завершен и ожидает завершения другого потока;

− INTERRUPT — поток блокирован и ожидает прерывания (при вызове функции InterruptWait());

− JOIN — поток блокирован и ожидает завершения другого потока (например, при вызове функции pthread_join());

− MUTEX — поток блокирован блокировкой взаимного исключения (например, при вызове функции pthread_mutex_lock());

− NANOSLEEP — поток находится в режиме ожидания в течение короткого периода времени (например, при вызове функции nanosleep());
− NET_REPLY — поток ожидает ответа на сообщение другого узла сети (поток вызвал функцию MsgReply);
− NET_SEND — поток ожидает получение импульса или сигнала от другого узла сети (поток вызвал функцию MsgSendPulse(), MsgDeliverEvent или SignalKill());
− READY — поток ожидает выполнения, пока процессор занят выполнением другого потока равного или более высокого приоритета;
− RECEIVE — поток блокирован на операции получения сообщения (поток вызвал функцию MsgReceive);
− REPLY — поток блокирован при ответе на сообщение (поток вызвал функцию MsgReply);
− RUNNING — поток выполняется процессором;
− SEM — поток ожидает освобождения семафора (поток вызвал функцию SyncSemWait());
− SEND — поток блокируется при отправке сообщения (поток вызвал функцию MsgSend);
− SIGSUSPEND — поток блокирован и ожидает сигнала (поток вызвал функцию sigsuspend());
− SIGWAITINFO — поток блокирован и ожидает сигнала (поток вызвал функцию sigwaitinfo());
− STACK — поток ожидает выделения виртуального адресного пространства для своего стека (родительский поток вызывает функцию ThreadCreate());
− STOPPED — поток блокирован и ожидает сигнала SIGCOUNT;
− WAITCTX — поток ожидает доступности нецелочисленного контекста;
− WAITPAGE — поток ожидает выделения физической памяти для виртуального адреса;
− WAITTHREAD — поток ожидает завершения создания дочернего потока (поток вызвал функцию ThreadCreate()).

6.2.3. Планирование потоков

При каждом вызове ядра, исключении или аппаратном прерывании, в результате которого управление передается коду ядра, выполнение активного потока временно приостанавливается. Действие планирования совершается в момент изменения состояния любого потока независимо от того, в каком процессе поток расположен. Планирование потоков осуществляется по всем процессам сразу [9].

Как правило, выполнение приостановленного потока через некоторое время возобновляется. При этом планировщик (sclleduler) выполняет переключение контекстов с одного потока на другой всякий раз, когда активный поток [9]:

– ***Блокируется.*** Активный поток блокируется, если он должен ожидать какого-либо события (например, ответа на запрос механизма обмена сообщениями, освобождения мьютекса и т. д.). Блокированный поток удаляется из очереди готовности, после чего запускается поток с наивысшим приоритетом. Когда блокированный поток разблокируется, он помещается в конец очереди готовности на соответствующий приоритетный уровень.

– ***Вытесняется.*** Активный поток вытесняется, когда поток с более высоким приоритетом помещается в очередь готовности (т. е. он переходит в состояние готовности (READY) в результате снятия условия блокировки). Прерванный поток остается на соответствующем приоритетном уровне в начале очереди готовности, а поток с более высоким приоритетом начинает выполняться.

– ***Отдает управление.*** Активный поток самостоятельно освобождает процессор и помещается в конец очереди готовности на данном уровне приоритета. После этого запускается поток с наивысшим приоритетом (в том числе им может быть поток, который только что отдал управление).

Каждому потоку назначается свой приоритет. Планировщик выбирает поток для выполнения в соответствии с приоритетом каждого потока, находящегося в состоянии готовности (READY), т. е. способного использовать процессор. Таким образом выбирается поток с наивысшим приоритетом [9].

Всего в ОС QNX Neutrino 6.3 поддерживается до 256 уровней приоритетов. Приоритет выполнения каждого непривилегированного потока может изменяться в пределах от 1 до 63 (наивысший приоритет), независимо от его дисциплины планирования. Только привилегированные (root) потоки могут иметь приоритет выше 63. Специальный поток с именем ***idle*** в администраторе процессов, имеет приоритет 0 и всегда готов к выполнению. По умолчанию поток наследует приоритет своего родительского потока [9].

Команда *procnto -p* позволяет изменить диапазон допустимых приоритетов для непривилегированного процесса:

procnto -p приоритет

Следует обратить внимание на то, что для предотвращения инверсии приоритетов ядро может временно повышать приоритет потока.

Потоки, находящиеся в очереди, упорядочиваются по приоритету. В действительности очередь готовых потоков состоит из 256 отдельных очередей по одной на каждый приоритет. Потоки выстраиваются в очереди по порядку поступления (FIFO) и в соответствии с их приоритетом. Для выполнения выбирается первый поток с наивысшим приоритетом [9].

Для работы в ОС QNX Neutrino 6.3 используются следующие алгоритмы планирования:

- FIFO-планирование (см. п. 3.1.2);
- Циклическое планирование (Round robin, см. п. 3.1.2);
- Спорадическое планирование.

Каждый поток в системе может выполняться по любому из методов. Методы применяются для каждого отдельного потока, а не для потоков или процессов одновременно.

FIFO-планирование и циклическое планирование применяется только в случаях, когда два или более потоков, имеющих одинаковый приоритет, находятся в состоянии готовности.

Спорадическое планирование. Алгоритм спорадического планирования обычно используется для задания верхнего лимита на время выполнения потока в пределах заданного периода времени. Этот метод необходим при выполнении монотонного частотного анализа системы, обслуживающей как периодиче-

ские, так и апериодические события. По сути, данный алгоритм позволяет потоку обслуживать апериодические события, не препятствуя своевременному выполнению других потоков или процессов в системе [9].

Как и в FIFO-планировании, поток, для которого применяется спорадическое планирование, выполняется до тех пор, пока он не блокируется или прерывается потоком с более высоким приоритетом. Кроме того, так же как и в адаптивном планировании, поток, для которого применяется спорадическое планирование, получает пониженный приоритет. Однако спорадическое планирование дает значительно более точное управление потоком [9].

При спорадическом планировании, приоритет потока может динамически изменяться между приоритетом переднего плана (Foreground) (или нормальным приоритетом) и фоновым (Background) (или пониженным) приоритетом. Для управления этим спорадическим переходом используются следующие параметры [9]:

– ***начальный бюджет потока (C)*** — количество времени, за которое поток может выполняться с нормальным приоритетом (N), перед тем как получить пониженный приоритет (L);

– ***пониженный приоритет (L)*** — приоритетный уровень, до которого приоритет потока будет снижен. При пониженном приоритете (L) поток выполняется в фоновом режиме. Если же поток имеет нормальный приоритет (N), он выполняется с приоритетом переднего плана;

– ***период пополнения (T)*** — период времени, в течение которого поток может расходовать свой бюджет выполнения. Для планирования операций пополнения в POSIX-стандартах это значение также используется в качестве сдвига по времени, отсчитываемого от того момента, когда поток переходит в состояние готовности (READY);

– ***максимальное число текущих пополнений*** — это значение устанавливает ограничение на количество выполняемых операций пополнения, тем самым ограничивая объем системных ресурсов, выделяемых на дисциплину спорадического планирования.

Как видно из рис. 6.3 [9], алгоритм спорадического планирования устанавливает начальный бюджет выполнения потока (C), который расходуется потоком в процессе его выполнения и пополняется с периодичностью, определенной параметром T. Когда поток блокируется, израсходованная часть бюджета выполнения потока (R) пополняется через какое-то установленное время (например, через 40 мс), отсчитываемое от момента, когда поток перешел в состояние готовности.

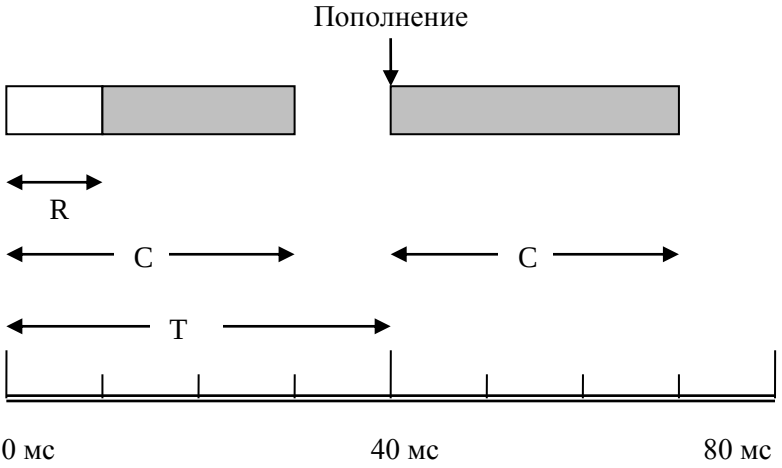

Рис. 6.3. Пример работы потока при спорадическом планировании

При нормальном приоритете (N) поток выполняется в течение периода времени, установленного его начальным бюджетом выполнения (C). После того как этот период истекает, приоритет потока снижается до пониженного уровня (L) до тех пор, пока не произойдет операция пополнения.

В этом случае поток перейдет на уровень с пониженным приоритетом (фоновый режим), на котором его выполнение будет зависеть от приоритета других потоков в системе.

Как только происходит пополнение, приоритет потока повышается до начального уровня. Таким образом, в правильно настроенной системе поток выполняется каждый период времени T в течение максимального времени C. Это обеспечивает такой порядок, при котором каждый поток, выполняемый с прио-

ритетом N, будет использовать только ***C/T процентов*** системных ресурсов.

Когда поток блокируется несколько раз, несколько операций пополнения могут происходить в разные моменты времени. Это может означать, что бюджет выполнения потока в пределах периода времени T дойдет до значения C; однако на протяжении этого периода бюджет может и не быть непрерывным [9].

Во время выполнения потока его приоритет может изменяться либо в результате прямого действия самого потока, либо в результате вмешательства ядра при получении сообщения от какого-либо потока с более высоким приоритетом [9].

Изменяться может не только приоритет, но и алгоритм планирования, применяемый ядром для выполнения потока. В таблице 6.2 приводится список POSIX-вызовов, используемых для управления потоком, а также соответствующие вызовы микроядра, используемые этими библиотечными функциями [9].

Таблица 6.2. POSIX-вызовы для управления потоком и вызовы микроядра

POSIX-вызов	**Вызов микроядра**	**Описание**
sched_getparam()	SchedGet()	Получить приоритет
sched_setparam()	SchedSet()	Установить приоритет
sched_getscheduler()	SchedGet()	Получить дисциплину планирования
sched_setscheduler()	SchedSet()	Установить дисциплину планирования

6.2.4. Управление потоками

Процесс может содержать один или несколько потоков. Число потоков варьируется. Один разработчик программного обеспечения, используя только единственный поток, может реализовать те же самые функциональные возможности, что и другой, используя пять потоков. Некоторые задачи сами по себе приводят к многопоточности и дают относительно простые решения, другие в силу своей природы, являются однопоточными,

и свести их к моногопоточночной реализации достаточно трудно.

Любой поток может создать другой поток в том же самом процессе. На это не налагается никаких ограничений (за исключением объема памяти). Как правило, для этого применяется функция POSIX *pthread_create()*:

```
#include <pthread.h>
int
pthread_create(pthread_t *thread,
                const pthread_attr_t *attr,
                void  *(*start_routine)  (void
*),
                void *arg);
```

thread — указатель на pthread_t, где храниться идентификатор потока;
attr — атрибутивная запись;
start_routine — подпрограмма с которой начинается поток;
arg — параметр, который передается подпрограмме start_routine.

Первые два параметра необязательные вместо них можно передавать NULL.

Параметр thread можно использовать для хранения идентификатора вновь создаваемого потока.

Пример однопоточной программы. Предположим, что мы имеет программу, выполняющую алгоритм трассировки луча. Каждая строка растра не зависит от остальных. Это обстоятельство (независимость строк растра) автоматически приводит к программированию данной задачи как многопоточной.

```
int main ( int argc, char **argv)
{
int x1;
… // Выполнить инициализации
for (x1 = 0; x1 < num_x_lines; x1++)
     {
```

```
      do_one_line (x1);
      }
… // Вывести результат
}
```

Здесь видно, что программа независимо по всем значениям x1 рассчитывает необходимые растровые строки.

Пример первой многопоточной программы. Для параллельного выполнения функции do_one_line (x1) необходимо изменить программу следующим образом:

```
int main ( int argc, char **argv)
{
int x1;
… // Выполнить инициализации
for (x1 = 0; x < num_x_lines; x1++)
      {
pthread_create (NULL, NULL, do_one_line,
                  (void *) x1);
      }
… // Вывести результат
}
```

Пример второй многопоточной программы. В приведенном примере непонятно когда нужно выполнять вывод результатов, так как приложение запустило массу потоков, но не знает когда они завершаться. Можно поставить задержку выполнения программы (sleep 1), но это не будет правильно. Для этого лучше использовать функцию pthread_join().

Есть еще один минус у приведенной выше программы, если у нас много строк в изображении не факт, что все созданные потоки будут функционировать параллельно, как правило, процессоров системе, гораздо меньше. Для этого лучше модифицировать программу так, чтобы запускалось столько потоков, сколько у нас процессоров в системе.

```
int num_lines_per_cpu;
```

```
int num_cpus;

int main (int argc, char **argv)
{
int cpu;
pthread_t *thread_ids;

… // Выполнить инициализации

// Получить число процессоров
num_cpus = _syspage_ptr->num_cpu;

thread_ids = malloc (sizeof (pthread_t) *
                                    num_cpus);
num_lines_per_cpu = num_x_lines / num_cpus;

for (cpu = 0; cpu < num_cpus; cpu++)
     {
     pthread_create (&thread_ids [cpu], NULL,
                     do_one_batch,
                     (void *) cpu);
     }

// Синхронизировать с завершением всех потоков
for (cpu = 0; cpu < num_cpus; cpu++)
     {
     pthread_join (thread_ids [cpu], NULL);
     }


… // Вывести результат
}

void *do_one_batch (void *c)
{
int cpu = (int) c;
int x1;
for (x1 = 0; x1 < num_lines_per_cpu; x1++)
```

```
        {
        do_one_line(x1        +        cpu        *
num_lines_per_cpu);
        }
}
```

6.3. Механизмы синхронизации

6.3.1. Перечень механизмов синхронизации

В ОС QNX используются POSIX-примитивы для синхронизации на уровне потоков. Некоторые из этих примитивов могут применяться для потоков в разных процессах. К службам синхронизации относятся объекты, приведенные в таблице 6.3 [9].

Таблица 6.3. Службы синхронизации

Служба синхронизации	Межзадачная поддержка	Сетевая поддержка
Мьютекс	Да	Нет
Условная переменная	Да	Нет
Барьер	Нет	Нет
Ждущая блокировка	Нет	Нет
Блокировка чтения/записи	Да	Нет
Семафор	Да	Да (только для именованных)
FIFO-планирование	Да	Нет
Отправка/получение/ответ	Да	Да
Атомарная операция	Да	Нет

6.3.2. Блокировки взаимного исключения (мьютексы)

Наиболее простым и из служб синхронизации являются мьютексы. Мьютекс (от англ. mutex — mutual exclusion lock) служит для обеспечения монопольного доступа к данным, которые совместно используются несколькими потоками. Операциями захвата мьютекса и освобождения мьютекса обычно обрам-

ляются участки кода, который обращается к совместно используемым данным (обычно это критическая секция кода) [9].

В каждый момент времени мьютекс может быть захвачен только одним потоком. Потоки, которые пытаются захватить уже захваченный мьютекс, блокируются до тех пор, пока этот мьютекс не освобождается. Когда поток освобождает мьютекс, поток с наивысшим приоритетом, который ожидает возможности захватить мьютекс, будет разблокирован и станет новым владельцем мьютекса. Таким образом, потоки обрабатывают критическую секцию кода в порядке своих приоритетов [9].

В большинстве процессоров захват мьютекса не требует обращения к ядру. Это достигается благодаря операции «сравнить и переставить» в семействе Intel x86, а также посредством условных инструкций «загрузить/сохранить» на большинстве процессоров семейства RISC [9].

При захвате мьютекса передача управления коду ядра происходит только при условии, что мьютекс уже захвачен и поток может быть включен в список блокированных потоков. Управление передается коду ядра также при освобождении мьютекса, если другие потоки ожидают разблокирования на этом мьютексе. Это позволяет выполнять захват и освобождение критических секций с очень высокой скоростью. Таким образом, действие ОС сводится всего лишь к управлению приоритетами [9].

Для определения текущего состояния мьютекса используется специальная неблокирующая функция *pthread_mutex_trylock()*. Для повышения производительности системы время выполнения критической секции кода должно быть небольшим и ограниченным. Если поток может блокироваться во время выполнения критической секции, то должна использоваться условная переменная [9].

Если поток, имеющий более высокий приоритет, чем владелец мьютекса, пытается захватить мьютекс, то действующий приоритет текущего владельца устанавливается равным приоритету блокированного потока, ожидающего мьютекс. Владелец вернется к своему исходному приоритету после того, как он освободит мьютекс. Такая схема не только обеспечивает мини-

мальное время ожидания мьютекса, но и решает классическую проблему инверсии приоритетов [9].

Функция *pthread_mutex_setrecursive()* позволяет изменять атрибуты мьютекса таким образом, чтобы один и тот же поток мог выполнять рекурсивный захват мьютекса. Это дает потоку возможность вызывать процедуры для выполнения повторного захвата мьютекса, который к этому моменту уже захвачен данным потоком [9].

6.3.3. Условные переменные

Условная переменная используется для блокировки потока по какому-либо условию во время выполнения критической секции кода. Условие может быть сколь угодно сложным и не зависит от условной переменной. Однако условная переменная всегда должна использоваться совместно с мьютексом для проверки условия [9].

Условные переменные поддерживают следующие функции [9]:

– ожидание условной переменной (*pthread_cond_wait()*);
– единичная разблокировка потока (*pthread_cond_signal()*)[3];
– множественная разблокировка потока (*pthread_cond_broadcast()*).

Приведем пример типичного использования условной переменной [9]:

```
pthread_mutex_lock ( &m );
...
while (!arbitrary condition) {
pthread_cond_wait( &cv, &m );
}
...
pthread_mutex_unlock ( &m );
```

В этом примере захват мьютекса происходит до проверки условия. Таким образом, проверяемое условие применяется

[3] Следует иметь в виду, что единичная разблокировка потока, обозначаемая термином «signal», никак не связана с понятием сигнала в стандартах POSIX.

только к текущему потоку. Пока данное условие является истинным, эта секция кода блокируется на вызове ожидания до тех пор, пока какой-либо другой поток не выполнит операцию единичной или множественной разблокировки потока по условной переменной [9].

Цикл *while* в приведенном примере требуется по двум причинам. Во-первых, стандарты POSIX не гарантируют отсутствие ложных пробуждений (например, в многопроцессорных системах). Во-вторых, если другой поток изменяет условие, необходимо заново выполнить его проверку, чтобы убедиться, что изменение соответствует принятым критериям. При блокировании ожидающего потока связанный с условной переменной мьютекс атомарно освобождается функцией *pthread_cond_wait()* для того, чтобы другой поток мог войти в критическую секцию программного кода [9].

Поток, который выполняет единичную разблокировку потока, разблокирует поток с наивысшим приоритетом, который стоит в очереди на условной переменной. Операция множественной разблокировки потока разблокирует все потоки, стоящие в очереди на условной переменной. Связанный с условной переменной мьютекс освобождается атомарно разблокированным потоком с наивысшим приоритетом. После обработки критической секции кода этот поток должен освободить мьютекс [9].

Другой вид операции ожидания условной переменной (*pthread_cond_timedwait()*) позволяет установить таймаут. По окончании этого периода ожидающий поток может быть разблокирован [9].

6.3.4. Барьеры

Барьер — это механизм синхронизации, который позволяет скоординировать работу нескольких взаимодействующих потоков (например, при матричных вычислениях) таким образом, чтобы каждый из них остановился в заданной точке в ожидании остальных потоков, прежде чем продолжить свою работу.

В отличие от функции *pthread_join()*, при которой поток ожидает завершения другого потока, барьер заставляет потоки

встретиться в определенной точке. После того как заданное количество потоков достигает установленного барьера, все эти потоки разблокируются и продолжат свою работу.

Барьер создается с помощью функции *pthread_barrier_init()*.

После создания барьера каждый поток вызывает функцию *pthread_barrier_wait()*, тем самым сообщая о завершения этого действия.

Когда поток вызывает функцию *pthread_barrier_wait()*, он блокируется до тех пор, пока то число потоков, которое было задано функцией *pthread_barrier_init()*, не вызовет функцию *pthread_barrier_wait()* и, соответственно, не заблокируется. После того как заданное количество потоков вызовет функцию *pthread_barrier_wait()*, все они разблокируются одновременно [9].

6.3.5. Ждущие блокировки

Ждущие блокировки работают аналогично условным переменным, за исключением некоторых деталей. Как и условные переменные, ждущие блокировки (*pthread_sleepon_lock()*) могут использоваться для блокировки потока до тех пор, пока условие не станет истинным (аналогично изменению значения ячейки памяти). Но в отличие от условных переменных (которые должны существовать для каждого проверяемого условия), ждущие блокировки применяются к одному мьютексу и динамически создаваемой условной переменной независимо от количества проверяемых условий. Максимальное число условных переменных в конечном итоге равно максимальному числу блокированных потоков. Этот вид блокировок аналогичен тем, которые применяются в ядре UNIX [9].

6.3.6. Блокировки по чтению/записи

Блокировки по чтению/записи (или более точное название «блокировки на множественное чтение и однократную запись») используются в тех случаях, когда доступ к структуре данных должен определяться по следующей схеме: чтение данных вы-

полняет множество потоков, запись — не более одного потока [9].

Блокировка по чтению/записи (*pthread_rwlock_rdlock()*) предоставляет доступ по чтению всем потокам, которые его запрашивают. Однако если поток запрашивает блокировку по записи (*pthread_rwlock_wrlock()*), запрос отклоняется до тех пор, пока все потоки, выполняющие чтение, не снимут свои блокировки по чтению (*pthread_rwlock_unlock()*)[9].

Множество потоков, выполняющих запись, выстраиваются в очередь (в порядке своих приоритетов), ожидая возможности выполнить операцию записи в защищенную структуру данных. Все блокированные потоки, выполняющие запись, запускаются до того, как читающие потоки снова получат разрешение на доступ к данным. Приоритеты читающих потоков не учитываются [9].

Существуют специальные вызовы, которые позволяют потоку тестировать возможность доступа к необходимой блокировке, оставаясь в активном состоянии (*pthread_rwlock_tryrdlock()* и *pthread_rwlock_trywrlock()*). Эти вызовы возвращают код завершения, сообщающий о возможности или невозможности установки блокировки [9].

Реализация блокировок по чтению/записи происходит не в ядре, а посредством мьютексов и условных переменных, предоставляемых ядром [9].

6.3.7. Семафоры

Еще одним средством синхронизации являются семафоры[4], которые позволяют потокам увеличивать (с помощью функции *sem_post()*) или уменьшать (с помощью функции *sem_wait()*) значение счетчика на семафоре для управления блокировкой потока (операции «post» и «wait» соответственно).

При вызове функции *sem_wait()* поток не блокируется, если счетчик имел положительное значение. При неположительном значении счетчика поток блокируется до тех пор, пока ка-

[4] Как правило, мьютексы работают быстрее, чем семафоры (которые всегда требуют обращения к коду ядра).

кой-либо другой поток не увеличит значение счетчика. Увеличение значения счетчика может выполняться несколько раз подряд, что позволяет уменьшать значение счетчика, не вызывая блокировки потоков.

Существенным отличием семафоров от других примитивов синхронизации является то, что семафоры безопасны для применения в асинхронной среде и могут управляться обработчиками сигналов. Семафоры как раз подходят для тех случаев, когда требуется пробудить поток с помощью обработчика сигнала.

Другим полезным свойством семафоров является то, что они были определены для работы между процессами. Хотя мьютексы тоже работают между процессами, стандарты POSIX рассматривают эту возможность как дополнительную функцию, которая может оказаться не переносимой между системами. Что касается синхронизации между потоками внутри одного процесса, то здесь мьютексы более эффективны, чем семафоры.

Полезной разновидностью семафоров является служба именованных семафоров. Эта служба использует администратор ресурсов и позволяет применять семафоры между процессами, выполняемыми на разных машинах внутри сети[5][9].

6.3.8. Синхронизация с помощью алгоритма планирования

Применение алгоритма FIFO-планирования стандарта POSIX в системе без симметричной многопроцессорной обработки предотвращает выполнение критической секции кода одновременно несколькими потоками с одинаковым приоритетом. Алгоритм FIFO-планирования предписывает, что все потоки, запланированные к выполнению по этому алгоритму и имеющие одинаковый приоритет, выполняются до тех пор, пока они самостоятельно не освободят процессор для другого потока [9].

Такое «освобождение» процессора также может произойти в случае, когда поток блокируется в результате обращения к

[5] Именованные семафоры работают медленнее, чем неименованные

другому процессу за сервисом или при получении сигнала. Поэтому критическая секция кода должна быть тщательно проработана и документирована для того, чтобы последующее обслуживание этого кода не приводило к нарушению данного условия [9].

Кроме того, потоки с более высоким приоритетом в том (или любом другом) процессе все же могут вытеснять потоки, выполняемые по алгоритму FIFO-планирования. Поэтому все потоки, которые могут «столкнуться» между собой внутри критической секции кода, должны планироваться по алгоритму FIFO с одинаковым приоритетом. При таком условии потоки могут обращаться к этой разделяемой памяти без необходимости делать предварительный явный запрос на синхронизацию [9].

6.3.9. Синхронизация с помощью механизма обмена сообщениями

Службы обмена сообщениями, используемые в ОС QNX, осуществляют неявную синхронизацию посредством блокировок. Во многих случаях они могут заменить собой другие службы синхронизации. Кроме того, службы обмена сообщениями — единственные примитивы синхронизации и межзадачного взаимодействия (кроме именованных семафоров, основанных на механизме обмена сообщениями), которые могут работать в сети [9].

6.3.10. Синхронизация с помощью атомарных операций

В некоторых случаях необходимо выполнить какую-либо небольшую операцию (например, увеличить значение переменной) *атомарно,* т. е. с гарантией того, что она не будет вытеснена другим потоком или каким-либо обработчиком прерываний.

В ОС QNX атомарные операции применяются для:

- сложения;
- вычитания;
- обнуления битов;

– установки битов;
– переключения битов.

Использоваться атомарные операции могут где угодно, но особенно они полезны в следующих двух случаях:

– при взаимодействии между обработчиком прерываний (ISR) и потоком;
– при взаимодействии между двумя потоками (как при многопроцессорной, так и при однопроцессорной обработке).

Поскольку обработчик прерываний может вытеснять поток в любой точке, единственным способом защиты потока от прерываний остается запрещение прерываний. Поскольку запрещение прерываний неприемлемо для системы реального времени, рекомендуется применение атомарных операций, которые предусмотрены в ОС QNX.

6.4. Межзадачное взаимодействие

6.4.1. Формы межзадачного взаимодействия

Благодаря реализованному в ОС QNX механизму межзадачного взаимодействия операционная система является полнофункциональной POSIX-совместимой системой. Механизм межзадачного взаимодействия соединяет между собой различные компоненты, добавляемые в микроядро для обеспечения системных служб, и объединяет их в единое целое.

В ОС QNX основной формой межзадачного взаимодействия является **обмен сообщениями**, но кроме него используются и несколько других форм. В большинстве случаев, все другие формы межзадачного взаимодействия надстраиваются поверх механизма обмена сообщениями ОС QNX. ***Основная стратегия здесь состоит в том, чтобы создать простую и надежную службу межзадачного взаимодействия, которую можно оптимизировать для максимальной производительности, используя минимум кода ядра.*** На основе этой службы потом можно создать более сложные механизмы межзадачного взаимодействия.

Высокоуровневые службы межзадачного взаимодействия (например, неименованные и именованные программные кана-

лы, реализованные на основе механизма обмена сообщениями ОС QNX) с аналогичными службами монолитного ядра, выдают приблизительно одинаковые характеристики производительности.

В ОС QNX Neutrino предусмотрены следующие формы межзадачного взаимодействия:

- обмен сообщениями (на уровне ядра);
- сигналы (на уровне ядра);
- очереди сообщений POSIX (внешний процесс);
- разделяемая память (на уровне администратора процессов);
- неименованные программные каналы (внешний процесс);
- именованные программные каналы (внешний процесс).

6.4.2. Связь между процессами посредством сообщений

Механизм передачи межпроцессных сообщений занимается пересылкой сообщений между процессами и является одной из важнейших частей операционной системы, так как все общение между процессами, в том числе и системными, происходит через сообщения. Сообщение в QNX – это последовательность байтов произвольной длины (0-65 535 байтов) произвольного формата. Протокол обмена сообщениями выглядит таким образом: задача блокируется для ожидания сообщения. Другая же задача посылает первой сообщение и при этом блокируется сама, ожидая ответа. Первая задача разблокируется, обрабатывает сообщение и отвечает, разблокируя при этом вторую задачу.

Сообщения и ответы, пересылаемые между процессами при их взаимодействии, находятся в теле отправляющего их процесса до того момента, когда они могут быть приняты. Это значит, что, с одной стороны, уменьшается вероятность повреждения сообщения в процессе передачи, а с другой – уменьшается объем оперативной памяти, необходимый для работы ядра. Кроме того, уменьшается число пересылок из памяти в память, что разгружает процессор. Особенностью процесса передачи сообщений является то, что в сети, состоящей из нескольких

компьютеров под управлением QNX, сообщения могут прозрачно передаваться процессам, выполняющимся на любом из узлов.

Для прямой связи друг с другом взаимодействующие процессы используют следующие функции:

– `Send()` — для посылки сообщений;
– `Receive()` — для приема сообщений;
– `Reply()` — для ответа процессу, пославшему сообщение.

Эти функции могут использоваться локально или по всей сети.

Обратите внимание на то, что для прямой связи процессов друг с другом необязательно использование функций `Send()`, `Receive()` и `Reply()`. Система библиотечных функций QNX надстроена над системой обмена сообщениями, поэтому процессы могут использовать передачу сообщений косвенно при использовании стандартных сервисных средств, например, программных каналов (pipe).

Приведем пример (рис. 6.4), который иллюстрирует использование функций `Send()`, `Receive()` и `Reply()` при взаимодействии двух процессов — A и B:

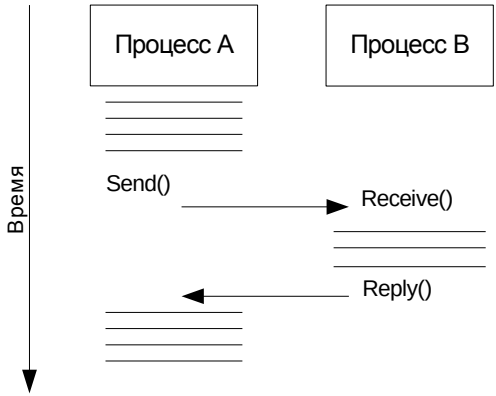

Рис. 6.4. Обмен сообщениями между процессами

– Процесс A посылает сообщение процессу B, выдав ядру запрос `Send()`. С этого момента процесс A становится SEND-

блокированным до тех пор, пока процесс B не выдаст `Receive()`, подтверждая получение сообщения;

– Процесс B выдает `Receive()` процессу A, ожидающему сообщения. Процесс A изменяет свое состояние на REPLY-блокированное. Поскольку от процесса B ожидается сообщение, он не блокируется.

Обратите внимание на то, что, если бы процесс B выдал `Receive()` до отправления ему сообщения, он оставался бы RECEIVE-блокированным до момента приема сообщения. В этом случае процесс A (отправитель) перешел бы сразу в REPLY-блокированное состояние после отправления сообщения процессу B;

Процесс B выполняет необходимую обработку, определяемую полученным от процесса A сообщением, и выдает `Reply()`. Процесс A получает ответное сообщение и разблокируется. Процесс B также разблокируется. Какой из процессов начнет выполняться первым, зависит от их относительных приоритетов.

Передача сообщений служит не только для обмена данными между процессами, но, кроме того, является средством синхронизации выполнения нескольких взаимодействующих процессов.

Давайте обратимся снова к приведенному выше рис. 6.4. После того, как процесс A выдаст запрос `Send()`, он не сможет выполняться до тех пор, пока не получит ответа на переданное им сообщение. Это служит гарантией того, что обработка данных, выполняемая процессом B для процесса A завершится до того, как процесс A сможет продолжить свою работу. В свою очередь процесс B после выдачи запроса `Receive()` может продолжать свою работу до поступления другого сообщения.

Рассмотрим более подробно функции `Send()`, `Receive()` и `Reply()`.

Использование функции `Send()`. Предположим, что процесс A выдает запрос на передачу сообщения процессу B. Запрос оформляется вызовом функции `MsgSend()`.

```
MsgSend (pid, smsg, rmsg, smsg_bn, rmsg_len);
```

Функция `MsgSend()` имеет следующие аргументы:

pid — идентификатор процесса-получателя сообщения (т.е. процесса B); pid — это идентификатор, посредством которого процесс опознается операционной системой и другими процессами;

smsg — буфер сообщения (т.е. посылаемого сообщения);

rmsg — буфер ответа (в который помещается ответ процесса B);

smsg_len — длина посылаемого сообщения;

rmsg_len — максимальная длина ответа, который должен получить процесс A.

Обратите внимание на то, что в сообщении будет передано не более чем smsg_len байт и принято в ответе не более чем rmsg_len байт — это служит гарантией того, что буферы никогда не будут переполнены.

Использование функции `Receive()`. Процесс B может принять запрос `MsgSend()`, выданный процессом A, с помощью функции `MsgReceive()`.

```
pid = MsgReceive (0, msg, msg_len);
```

Функция `MsgReceive()` имеет следующие аргументы:

pid — идентификатор процесса, пославшего сообщение (т.е. процесса A);

0 — (ноль) указывает на то, что процесс B готов принять сообщение от любого процесса;

msg — буфер, в который будет принято сообщение;

msg_len — максимальное количество байт данных, которое может поместиться в приемном буфере.

В том случае, если значения smsg_len в функции `MsgSend()` и msg_len в функции `MsgReceive()` различаются, то количество передаваемых данных будет определяться наименьшим из них.

Использование функции `Reply()`. После успешного приема сообщения от процесса A процесс B должен ответить ему, используя функцию `MsgReply()`.

```
MsgReply (pid, reply, reply_len)
```

Функция `MsgReply()` имеет следующие аргументы:

pid — идентификатор процесса, которому направляется ответ (т.е. процесса А);

reply — буфер ответа;

reply_len — длина сообщения, передаваемого в ответе.

Если значения reply_len в функции `MsgReply()` и rmsg_len в функции `MsgSend()` различаются, то количество передаваемых данных определяется наименьшим из них.

Пример передачи сообщений, который только что был рассмотрен, иллюстрирует наиболее типичный способ передачи, при котором обслуживающий процесс находится в RECEIVE-блокированом состоянии, ожидая запроса от другого процесса на выполнение какой-либо работы. Этот способ передачи сообщений называется ***Send-управляемым***, при котором процесс, требующий обслуживания, инициирует работу, посылая сообщение, а обслуживающий процесс завершает работу, выдавая ответ на принятое сообщение.

Существует еще и другой, менее распространенный, чем Send-управляемый, но в отдельных случаях более предпочтительный способ передачи сообщений, а именно ***Reply-управляемый***, при котором работа инициируется функцией `Reply()`. В соответствии с этим способом «рабочий» процесс посылает сообщение обслуживающему процессу, указывая на то, что он готов к работе. Обслуживающий процесс фиксирует, что «рабочий» процесс послал ему сообщение, но не отвечает ему немедленно. Через некоторое время обслуживающий процесс может ответить «рабочему» процессу. «Рабочий» процесс выполняет свою работу, а затем, завершив ее, посылает обслуживающему процессу сообщение, содержащее результаты.

Данные, передаваемые в сообщении, находятся в процессе-отправителе до тех пор, пока получатель не будет готов к обработке сообщения. Сообщение не копируется в ядро. Это обеспечивает сохранность данных, так как процесс-отправитель остается SEND-блокированным и не может случайным образом модифицировать данные сообщения.

При выдаче запроса `Reply()` данные, содержащиеся в ответном сообщении, передаются от отвечающего процесса

REPLY-блокированному процессу за одну операцию. Функция `Reply()` не блокирует отвечающий процесс, так как REPLY-блокированный процесс разблокируется сразу после того, как данные скопируются в его адресное пространство.

Процессу-отправителю нет никакой необходимости «знать» что-либо о состоянии процесса-получателя, которому он посылает сообщение. В том случае, если процесс-получатель будет не готов к приему сообщения, то процесс-отправитель после отправления сообщения просто перейдет в SEND-блокированное состояние.

При необходимости любой процесс может посылать сообщение нулевой длины, ответ нулевой длины, либо то и другое.

С точки зрения разработчика выдача запроса `Send()` обслуживающему процессу — это практически то же самое, что и обращение к библиотеке подпрограмм. В обоих случаях разработчик формирует некоторые наборы данных, а затем выдает `Send()` или обращается к библиотечной функции. После этого, между двумя определенными точками программы `Receive()` и `Reply()` — в одном случае, либо между входом функции и оператором завершения функции — `return` в другом, управление передается сервисным программам, при этом ваша программа ожидает завершения их выполнения. После того как сервисные программы отработали, ваша программа «знает», где находятся результаты их работы, и может затем анализировать коды ошибок, обрабатывать результаты и т.д.

Несмотря на это кажущееся сходство, процесс передачи сообщения намного сложнее обычного вызова библиотечной функции. Например, `Send()` может по сети обратиться к другой машине, где действительно будет выполняться сервисная программа. Кроме того, может быть организована параллельная обработка данных без создания нового процесса. Обслуживающий процесс выдает `Reply()`, позволяя вызывающему процессу продолжать выполняться, и затем продолжает свою работу.

Несколько процессов могут посылать сообщения одному процессу. Обычно процесс-получатель принимает сообщения в порядке их поступления, однако, может быть установлен режим

приема сообщений в порядке приоритетов процессов-отправителей.

Дополнительные возможности передачи сообщений. В системе QNX имеются функции, предоставляющие дополнительные возможности передачи сообщений, а именно:

– условный прием сообщений;
– чтение и запись части сообщения;
– передача составных сообщений.

Обычно для приема сообщения используется функция `Receive()`. Этот способ приема сообщений в большинстве случаев является наиболее предпочтительным.

Однако иногда процессу требуется предварительно «знать», было ли ему послано сообщение, чтобы не ожидать поступления сообщения в RECEIVE-блокированном состоянии. Например, процессу требуется обслуживать несколько высокоскоростных устройств, не способных генерировать прерывания и, кроме того, процесс должен отвечать на сообщения, поступающие от других процессов. В этом случае используется функция `Creceive()`, которая считывает сообщение, если оно становится доступным, или немедленно возвращает управление процессу, если нет ни одного отправленного сообщения.

ВНИМАНИЕ. По возможности следует избегать использования функции `Creceive()`, так как она позволяет процессу непрерывно загружать процессор на соответствующем приоритетном уровне.

В некоторых случаях предпочтительнее считывать или записывать только часть сообщения для того, чтобы использовать буфер, уже выделенный для сообщения, и не заводить рабочий буфер.

Например, администратор ввода/вывода может принимать для записи сообщения, состоящие из заголовка фиксированной длины и переменного количества данных. Заголовок содержит значение количества байт данных (от 0 до 64 Кбайт). Администратор ввода/вывода может принимать сначала только заголовок, а затем, используя функцию `Readmsg()`, считывать данные переменной длины в соответствующий буфер. Если количе-

ство посылаемых данных превышает размер буфера, администратор ввода/вывода может вызывать функцию `Readmsg()` несколько раз, передавая данные по мере освобождения буфера. Аналогично, функцию `Writemsg()` можно использовать для сбора и копирования данных в буфер отправителя по мере его освобождения, снижая таким образом требования к размеру внутреннего буфера администратора ввода/вывода.

До сих пор рассматривались сообщения представляемые как единый пакет байтов. Однако, как правило, сообщения состоят из нескольких дискретных частей. Например, сообщение может иметь заголовок фиксированной длины, за которым следуют данные переменной длины. Для того чтобы части сообщения эффективно передавались и принимались без копирования во временный рабочий буфер, составное сообщение может формироваться в нескольких раздельных буферах. Этот метод позволяет администраторам ввода/вывода системы QNX Dev и Fsys, обеспечивать высокую производительность.

Для работы с составными сообщениями используются следующие функции:

- `Creceivemx()`
- `Readmsgmx()`
- `Receivemx()`
- `Replymx()`
- `Sendmx()`
- `Writemsgmx()`

Все сообщения в системе QNX начинаются с 16-битового слова, которое называется кодом сообщения. Системные процессы QNX используют следующие коды сообщений:
0X0000 — 0X00FF — сообщения Администратора процессов;
0X0100 — 0X01FF — сообщения ввода/вывода (для всех обслуживающих программ);
0X0200 — 0X02FF — сообщения Администратора файловой системы;
0X0300 — 0X03FF — сообщения Администратора устройств;
0X0400 — 0X04FF — сообщения Сетевого администратора;
0X0500 — 0X0FFF — зарезервировано для системных процессов, которые могут появиться в будущем.

Каналы и соединения. В ОС QNX Neutrino сообщения передаются в направлении каналов и соединений, а не напрямую от потока к потоку. Поток, которому необходимо получить сообщение, сначала создает канал, а другой поток, которому необходимо передать сообщение этому потоку, должен сначала установить соединение, «подключившись» к этому каналу.

Каналы требуются для вызовов микроядра, предназначенных для обмена сообщениями, и используются серверами для приема сообщений с помощью функции `MsgReceive()`. Соединения создаются потоками-клиентами для того, чтобы «присоединиться» к каналам, открытым серверами. После того как соединения установлены, клиенты могут передавать по ним сообщения с помощью функции `MsgSend()`. Если несколько потоков процесса подключается к одному и тому же каналу, тогда для повышения эффективности все эти соединения отображаются в один объект ядра. Каналы и соединения, созданные процессом, обозначаются небольшими целочисленными идентификаторами. Клиентские соединения отображаются непосредственно в дескрипторы файлов [9] (рис. 6.5).

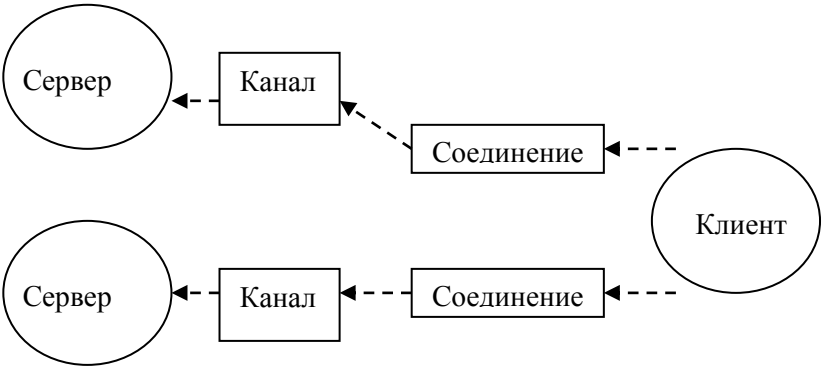

Рис. 6.5. Соединения по каналам клиента с сервером

С точки зрения архитектуры это имеет ключевое значение. Благодаря тому, что клиентские соединения отображаются непосредственно в дескриптор файла, на одну операцию преоб-

разования становится меньше. Отпадает необходимость «выяснять» из дескриптора файла, куда именно сообщение должно быть передано. Вместо этого сообщение передается непосредственно в «дескриптор файла» (т. е. идентификатор соединения). На канале может выстраиваться в очереди множество клиентов.

Функции для каналов и соединений приведены в таблице 6.4.

Таблица 6.4. Функции для каналов и соединений

Функции	**Описание**
ChannelCreate()	Создать канал для получения сообщений
ChannelDestroy()	Уничтожить канал
ConnectAttach()	Создать соединение для передачи сообщений
ConnectDetach()	Закрыть соединение

Импульсы. Кроме служб синхронизации `Send/Receive/Reply`, в ОС QNX Neutrino используются неблокирующие сообщения фиксированного размера. Эти сообщения называются импульсами (pulses) и имеют небольшую длину (4 байта данных и 1 байт кода).

Таким образом, импульсы несут в себе всего лишь 8 битов кода и 32 бита данных. Этот вид сообщений часто используется в качестве механизма уведомления внутри обработчиков прерываний. Импульсы также позволяют серверам не блокируясь передавать сообщения о событиях [9].

Наследование приоритетов. Серверный процесс получает сообщения в порядке приоритетов. После получения запроса потоки внутри сервера наследуют приоритет потока отправителя (но не алгоритм планирования). В результате относительные приоритеты потоков, осуществляющих запрос к серверу, сохраняются прежними, а сервер работает с соответствующим приоритетом. Таким образом, механизм наследования приоритетов на основе передачи сообщений позволяет избежать проблемы инверсии приоритетов [9].

События. Важным новшеством в архитектуре микроядра ОС QNX является подсистема обработки событий. Стандарты POSIX и их расширения реального времени определяют несколько асинхронных методов уведомления — например, UNIX-сигналы (не выстраивают данные в очередь и не пересылают их), POSIX-сигналы реального времени (могут выстраивать данные в очередь и пересылать их) и т.д.

ОС QNX обеспечивает очень гибкую архитектуру, в которой предусмотрен вызов ядра `MsgDeliverEvent()`, предназначенный для передачи неблокирующих событий. Все известные асинхронные службы могут реализовываться с использованием этой функции.

Программный интерфейс механизма обмена сообщениями состоит из функций, приведенных в таблице 6.5.

Таблица 6.5. Функции механизма обмена сообщениями

Функции	**Описание**
MsgSend()	Отправить сообщение и блокировать поток до получения ответа
MsgReceive()	Ожидать сообщение
MsgReceivePulse()	Ожидать импульс
MsgReply()	Ответить на сообщение
MsgError()	Переслать ответ содержащий статус потока
MsgRead()	Прочитать дополнительные данные из полученного сообщения
MsgWrite()	Записать дополнительные данные в ответное сообщение
MsgInfo()	Получить информацию о полученном сообщении
MsgSendPulse()	Передать импульс
MsgDeliverEvent()	Передать событие клиенту
MsgKeyData()	Снабдить сообщение ключом безопасности

6.4.3. Примеры связи между процессами посредствам обмена сообщениями

Клиент

Передача сообщения со стороны клиента осуществляется применением какой-либо функции из семейства *MsgSend()*.

Мы рассмотрим это на примере простейшей из них — *MsgSend():*

```
include <sys/neutrino.h>
int MsgSend(int coid, const void *smsg, int
sbytes,
                void *rmsg, int rbytes);
```

Для создания установки соединения между процессом и каналом используется функция *ConnectAttach()*, в параметрах которой задаются идентификатор процесса и номер канала.

```
#include <sys/neutrino.h>
int ConnectAttach( uint32_t nd,
                   pid_t pid,
                   int chid,
                   unsigned index,
                   int flags );
```

Пример:

Передадим сообщение процессу с идентификатором 77 по каналу 1:

```
#include <sys/neutrino.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

char *smsg = "Это буфер вывода";
char rmsg[200];
int coid;
int main(void);
```

```
{
// Установить соединение
     coid = ConnectAttach(0, 77, 1, 0, 0);
     if (coid == -1)
     {
      fprintf(stderr, "Ошибка ConnectAttach к
                       0/77/1!\n");
      perror(NULL);
      exit(EXIT_FAILURE);
     }
// Послать сообщение
     if(MsgSend(coid, smsg,
strlen(smsg)+1,rmsg,
                sizeof(rmsg)) == -1)
     {
      fprintf(stderr, "Ошибка MsgSendNn");
      perror(NULL);
      exit(EXIT FAILURE);
     }
     if (strlen(rmsg) > 0)
     {
      printf("Процесс   с   ID   77   возвратил
\"%s\"\n",
                rmsg);
     }
     exit(0);
}
```

Предположим, что процесс с идентификатором 77 был действительно активным сервером, ожидающим сообщение именно такого формата по каналу с идентификатором 1.

После приема сообщения сервер обрабатывает его и в некоторый момент времени выдает ответ с результатами обработки. В этот момент функция MsgSend() должна возвратить ноль (0), указывая этим, что все прошло успешно.

Если бы сервер послал нам в ответ какие-то данные, мы смогли бы вывести их на экран с помощью последней строки в

программе (с тем предположением, что обратно мы получаем корректную ASCIIZ-строку).

Сервер

Создание канала. Сервер должен создать канал — то, к чему присоединялся клиент, когда вызывал функцию *ConnectAttach()*.

Обычно сервер, однажды создав канал, приберегает его «впрок». Канал создается с помощью функции *ChannelCreate()* и уничтожается с помощью функции *ChannelDestroy():*

```
#include <sys/neutrino.h>
int ChannelCreate(unsigned flags);
int ChannelDestroy(int chid);
```

Пока на данном этапе будем использовать для параметра flags значение 0 (ноль). Таким образом, для создания канала сервер должен сделать так:

```
int  chid;
chid = ChannelCreate (0);
```

Теперь у нас есть канал. В этом пункте клиенты могут подсоединиться (с помощью функции *ConnectAttach()*) к этому каналу и начать передачу сообщений. Сервер отрабатывает схему сообщений обмена в два этапа — этап «приема» (receive) и этап «ответа» (reply).

```
#include <sys/neutrino.h>
int MsgReceive(int chid, void *rmsg, int
rbytes,
                 struct _msg_info *info);
int MsgReply(int rcvid, int status, const void
                 *msg, int nbytes);
```

Для каждой буферной передачи указываются два размера (в случае запроса от клиента это sbytes на стороне клиента и rbytes на стороне сервера; в случае ответа сервера это sbytes на

стороне сервера и rbytes на стороне клиента). Это сделано для того, чтобы разработчики каждого компонента смогли определить размеры своих буферов — из соображений дополнительной безопасности.

В нашем примере размер буфера функции MsgSend() совпадал с длиной строки сообщения.

Пример:

```
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <errno.h>
#include <stdio.h>
#include <process.h>
void main(void)
{
int  rcvid;
int   chid;
char  message [512];

// Создать канал
chid = ChannelCreate(0);

// Выполняться вечно — для сервера это обычное
дело
while (1)
{
      // Получить и вывести сообщение
      rcvid=MsgReceive(chid, message,
                        sizeof(message), NULL);
      printf ("Получил сообщение, rcvid %X\n",
                  rcvid);
      printf ("Сообщение такое: \"%s\",\n",
                  message);

// Подготовить ответ — используем тот же буфер
      strcpy (message, "Это ответ");
      MsgReply (rcvid, EOK, message,
```

```
                sizeof (message));
}
}
```

Как видно из программы, функция MsgReceive() сообщает ядру том, что она может обрабатывать сообщения размером вплоть до sizeof(message) (или 512 байт).

Наш клиент (представленный выше) передал только 28 байт (длина строки).

Определение идентификаторов узла, процесса и канала (ND/PID/CHID) нужного сервера

Для соединения с сервером функции ConnectAttach() необходимо указать дескриптор узла (Node Descriptor — ND), идентификатор процесса (process ID — PID), а также идентификатор канала (Channel ID — CHID).

Если один процесс создает другой процесс, тогда это просто — вызов создания процесса возвращает идентификатор вновь созданного процесса. Создающий процесс может либо передать собственные PID и CHID вновь созданному процессу в командной строке, либо вновь созданный процесс может вызвать функцию ***getppid()*** для получения идентификатора родительского процесса, и использовать некоторый «известный» идентификатор канала.

```
#include <process.h>
pid_t getpid( void );
```

Вопрос: «Как сервер объявляет о своем местонахождении?»

Существует множество способов сделать это; мы рассмотрим только три из них, в порядке возрастания «элегантности»:

1. Открыть файла с известным именем и сохранить в нем ND/PID/CHID. Такой метод является традиционным для серверов UNIX, когда сервер открывает файл (например, /etc/httpd.pid), записывает туда свой идентификатор процесса в виде строки ASCII и предполагают, что клиенты откроют этот файл, прочитают из него идентификатор.

2. Использовать для объявления идентификаторов ND/PID/ CHID глобальные переменные. Такой способ обычно применяется в многопоточных серверах, которые могут посылать сообщение сами себе. Этот вариант по самой своей природе является очень редким.

3. Занять часть пространства имен путей и стать администратором ресурсов.

6.4.4. Связь между процессами посредством сигналов

Связь посредством сигналов представляет собой традиционную форму асинхронного взаимодействия, используемую в различных операционных системах.

В системе QNX поддерживается набор POSIX-совместимых сигнала и специальные QNX-сигналы. Нумерация обоих наборов сигналов организована на основе набора из 64 однотипных сигналов, реализованных в ядре:

- 1..57 — 57 сигналов стандарта POSIX;
- 41..56 — 16 сигналов реального времени стандарта POSIX;
- 57..64 — 8 специальных сигналов ОС QNX.

Хотя сигналы реального времени в стандарте POSIX отличаются от UNIX-сигналов (во-первых, тем, что они могут содержать 4 байта данных и 1 байт кода, и, во-вторых, тем, что их можно ставить в очередь на передачу), каждый из них можно использовать по отдельности, а комбинированный набор сигналов всегда будет соответствовать стандарту POSIX [9].

Сигнал выдается процессу при наступлении некоторого заранее определенного для данного сигнала события. Процесс может выдать сигнал самому себе.

Если вы хотите сгенерировать сигнал из интерпретатора Shell, используйте утилиты kill или slay.

Если вы хотите сгенерировать сигнал из процесса, используйте утилиты kill() или raise().

В зависимости от того, каким образом был определен способ обработки сигнала, возможны три варианта его приема:

1. Если процессу не предписано выполнять каких-либо специальных действия по обработке сигнала, то по умолчанию поступление сигнала прекращает выполнение процесса;

2. Процесс может проигнорировать сигнал. В этом случае выдача сигнала не влияет на работу процесса (обратите внимание на то, что сигналы `SIGCONT`, `SIGKILL` и `SIGSTOP` не могут быть проигнорированы при обычных условиях);
3. Процесс может иметь обработчик сигнала, которому передается управление при поступлении сигнала. В этом случае говорят, что процесс может «ловить» сигнал. Фактически такой процесс выполняет обработку программного прерывания. Данные с сигналом не передаются.

Интервал времени между генерацией и выдачей сигнала называется задержкой. В данный момент времени для одного процесса могут быть задержаны несколько разных сигналов. Сигналы выдаются процессу тогда, когда планировщик ядра переводит процесс в состояние готовности к выполнению. Порядок поступления задержанных сигналов не определен.

Для задания способа обработки сигнала следует воспользоваться функцией ANSI C `signal()` или функцией POSIX `sigaction()`.

Функция `sigaction()` предоставляет больше возможностей по управлению средой обработки сигнала.

Способ обработки сигнала можно изменить в любое время. Если установить обработчику режим игнорирования сигналов, то все задержанные сигналы будут немедленно отменены.

Отметим некоторые особенности работы процессов, которые «ловят» сигналы с помощью обработчика сигналов.

Обработчик сигналов аналогичен программному прерыванию. Он выполняется асинхронно с другими программами процесса. Следовательно, обработчик сигналов может быть запущен во время выполнения любой функции в программе (включая библиотечные функции).

Если процессу не требуется возврата управления от обработчика сигналов в прерванную точку, то в этом случае в обработчике сигналов может быть использована функция `siglongjmp()` или `longjmp()`. Причем `siglongjmp()` предпочтительнее, т.к. в случае использования `longjmp()` сигнал остается блокированным.

Иногда может потребоваться временно задержать выдачу сигнала, не изменяя при этом способа его обработки. В системе QNX имеется набор функций, которые позволяют блокировать выдачу сигналов. После разблокировки сигнал выдается программе.

Во время работы обработчика сигналов QNX автоматически блокирует обрабатываемый сигнал. Это означает, что не требуется организовывать вложенные вызовы обработчика сигналов. Каждый вызов обработчика сигналов не прерывается остальными сигналами данного типа. При нормальном возврате управления от обработчика, сигнал автоматически разблокируется.

Для повышения эффективности передачи сигналов процессам, ядро выполняет кэширование последнего потока, принявшего сигнал, и впоследствии всегда передает сигнал этому потоку в первую очередь.

ВНИМАНИЕ. В некоторых версиях системы UNIX работа с обработчиком сигналов организована некорректно, так как в них не предусмотрена блокировка сигналов. В результате в некоторых приложениях, работающих под управлением UNIX, используется функция `signal()` *внутри обработчика прерываний с целью «перевзвода» обработчика. В этом случае может возникнуть одна из двух аварийных ситуаций. Во-первых, если другой сигнал поступает, во время работы обработчика, но вызова функции* `signal()` *еще не было, то программа будет снята с обработки. Во-вторых, если сигнал поступает сразу же после вызова обработчиком функции* `signal()`*, то обработчик будет запускаться рекурсивно. В QNX выполняется блокировка сигналов, поэтому указанные выше проблемы не могут возникнуть. Нет необходимости вызывать* `signal()` *из обработчика. Если требуется выйти из любой точки обработчика, то следует воспользоваться функцией* `siglongjmp()`.

Существует важная взаимосвязь между сигналами и сообщениями. Если при генерации сигнала ваш процесс окажется SEND-блокированным или RECEIVE-блокированным (причем

имеется обработчик сигналов), то будут выполняться следующие действия:

1) процесс разблокировывается;
2) выполняется обработка сигнала;
3) функции `Send()` или `Receive()` возвращают управление с кодом ошибки.

Если процесс был SEND-блокированным, то проблемы не возникает, так как получатель не получит сообщение. Но если процесс был REPLY-блокированным, то неизвестно, было, обработано отправленное сообщение или нет, а, следовательно, неизвестно, нужно ли еще раз выдавать `Send()`.

Процесс, выполняющий функции сервера (т.е. принимающий сообщения), может запрашивать уведомления о том, когда обслуживаемый процесс выдаст сигнал, находясь в REPLY-блокированном состоянии. В этом случае обслуживаемый процесс становится SIGNAL-блокированным с задержанным сигналом, и обслуживающий процесс принимает специальное сообщение, описывающее тип сигнала. Обслуживающий процесс может выбрать одно из следующих действий:

– нормально завершить первоначальный запрос: отправитель будет уведомлен о том, что сообщение было обработано надлежащим образом;
– освободить все закрепленные ресурсы и возвратить управление с кодом ошибки, указывающим на то, что процесс был разблокирован сигналом: отправитель получит чистый код ошибки.

Когда обслуживающий процесс сообщает другому процессу, что он SIGNAL-блокирован, сигнал выдается немедленно после возврата управления функцией `Send()`.

ОС QNX расширяет механизмы передачи сигналов стандарта POSIX благодаря тому, что позволяет направлять сигналы отдельным потокам, а не всему процессу, содержащему их. Поскольку сигналы — это асинхронные события, они также реализуются посредством механизмов передачи событий. В таблице 6.6 приведены вызовы микроядра и соответствующие POSIX-вызовы.

Таблица 6.6. Вызовы микроядра и соответствующие POSIX-вызовы

POSIX-вызов	**Вызов микроядра**	**Описание**
kill(), pthread_kill(), raise(), sigqueue()	SignalKill()	Установить сигнал на группе процессов, процессе или потоке
sigaction()	SignalAction()	Определить действие, выполняемое при получении сигнала
Sigprocmask(), pthread_sigmask()	SignalProcmask()	Изменить маску сигналов потока
sigsuspend(), pause()	SignalSuspend()	Блокироваться до тех пор, пока сигнал не вызовет обработчик сигнала
sigwaitinfo()	SignalWaitinfo()	Ожидать сигнала. После получения сигнала вернуть информацию о нем

Приведем краткое описание некоторых сигналов:

- SIGABRT — сигнал аварийного завершения;
- SIGALRM — сигнал таймаута;
- SIGCHLD — сигнал о завершении порожденного процесса;
- SIGCONT — продолжить выполнение, если процесс находится в состоянии HELD (задержан);
- SIGILL — некорректная команда;
- SIGINT — интерактивный предупредительный сигнал
- SIGKILL — сигнал завершения;
- SIGSYS — некорректный параметр системного вызова;
- SIGTERM — сигнал завершения;
- SIGTSTP — сигнал остановки, генерированный с клавиатуры;
- SIGUSR1 — зарезервирован как определяемый приложением сигнал 1;
- SIGUSR2 — зарезервирован как определяемый приложением сигнал 2;

– SIGWINCH — изменение размера окна.

6.5. Управление таймером

В QNX управление временем основано на использовании системного таймера. Этот таймер содержит текущее координатное универсальное время (UTC) относительно 0 часов 0 минут 0 секунд 1 января 1970 г. Для установки местного времени функции управления временем используют переменную среды TZ.

Программы интерпретатора Shell и процессы могут быть задержаны на заданное количество секунд с помощью простой утилиты таймирования. Программы интерпретатора используют для этого утилиту sleep; процессы — функцию Си `sleep()`. Можно также воспользоваться функцией `delay()`, в которой задается интервал времени в миллисекундах.

Процесс может также создавать таймеры, задавать им временной интервал и удалять таймеры. Эти более сложные средства таймирования соответствуют стандарту POSIX 1003.4/Draft 9.

Процесс может создать один или несколько таймеров. Таймеры могут быть любого поддерживаемого системой типа, а их количество ограничивается максимально допустимым количеством таймеров в системе.

Для создания таймера используется функция Си `mktimer()`. Эта функция позволяет задавать следующие типы механизма ответа на события:

– перейти в режим ожидания до завершения. Процесс будет находиться в режиме ожидания начиная с момента установки таймера до истечения заданного интервала времени;
– оповестить с помощью proxy. Proxy используется для оповещения процесса об истечении времени ожидания;
– оповестить с помощью сигнала. Сформированный пользователем сигнал выдается процессу по истечении времени ожидания.

Вы можете задать таймеру следующие временные интервалы:

– *абсолютный.* Время относительно 0 часов, 0 минут, 0 секунд, 1 января 1970 г.;
– *относительный.* Время относительно значения текущего времени.

Можно также задать повторение таймера на заданном интервале. Например, вы установили таймер на 9 утра завтрашнего дня. Его можно установить так, чтобы он срабатывал каждые пять минут после истечения этого времени. Можно также установить новый временной интервал существующему таймеру. Результат этой операции зависит от типа заданного интервала:
– для абсолютного таймера новый интервал замещает текущий интервал времени;
– для относительного таймера новый интервал добавляется к оставшемуся временному интервалу.

Для установки абсолютного временного интервала, используйте функцию `abstimer()`.

Для установки относительного временного интервала, используйте функцию `reltimer()`.

Для удаления таймера воспользуйтесь функцией Си `rmtimer()`. Таймер может удалить сам себя по истечении временного интервала при условии:
– при вызове `rmtimer()` включена опция _TNOTIFY_SLEEP;
– таймер неповторяемый.

Период таймера задается утилитой ticksize или функцией Си `qnx_timerperiod()`. Вы можете выбрать период в интервале от 500 микросекунд до 50 миллисекунд.

Для определения оставшегося времени таймирования или для того, чтобы узнать, был ли таймер удален, используйте функцию Си `gettimer()`.

Пример 1. Данная программа запускает в цикле ожидания на 10 секунд и прерывает это ожидание с помощью сигнала:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/siginfo.h>
#include <sys/neutrino.h>
```

```
#include <signal.h>
#include <time.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
struct itimerspec timer; // структура с описа-
нием
                         //таймера
timer_t timerid;// ID таймера

extern void handler(); //Обработчик таймера
struct sigaction act;  //Структура описывающая
                      //действие
                      //по сигналу
sigset_t set; //Набор сигналов нам необходимый
            //для таймера

sigemptyset( &set ); //Обнуление набора
sigaddset( &set, SIGALRM); //Включение в набор
                          //сигнала
                           //от таймера
act.sa_flags = 0;
act.sa_mask = set;
act.sa_handler = &handler; // Вешаем обработ-
чик
                          // на действие
sigaction( SIGALRM, &act, NULL); //Зарядить
сигнал,
                      // присваивание структу-
                      ры
                      // для конкретного сиг-
                 нала
                      // (имя сигнала,
                      // структура-действий)
```

```
//Создать таймер
if (timer_create (CLOCK_REALTIME, NULL,
&timerid)
                                        == -1)
{
  fprintf (stderr, "%s: ne udalos timer %d\n",
                                  "TM", errno);
}


// Данный макрос для сигнала SIGALRM не нужен
// Его необходимо вызывать
// для пользовательских сигналов
// SIGUSR1 или SIGUSR2. Функция timer_create()
// в качестве второго параметра должна
// использовать &event.

// SIGEV_SIGNAL_INIT(&event, SIGALRM);

timer.it_value.tv_sec=  3; //Взвести таймер
                          //на 3 секунды
timer.it_value.tv_nsec= 0;
timer.it_interval.tv_sec=  3;
     //Перезагружать
                             //таймер
timer.it_interval.tv_nsec= 0; // через 3 се-
кунды

timer_settime (timerid, 0, &timer, NULL);
                             //Включить таймер

for (;;)
{
    sleep(10);  // Спать десять секунд.
                // Использование таймера за-
          держки
    printf("More time!\n");
}
```

```
exit(0);
}

void handler( signo )
{
     //Вывести сообщение в обработчике.
     printf( "Alarm clock ringing!!!.\n");
     // Таймер заставляет процесс проснуться.
}
```

Здесь *it_value* и *it_interval* принимает одинаковые значения. Такой таймер сработает один раз (с задержкой *it_value*), а затем будет циклически перезагружаться с задержкой *it_interval*.

Оба параметра *it_value* и *itinterval* фактически являются структурами типа ***struct timespec*** — еще одного POSIX-объекта. Эта структура позволяет вам обеспечить разрешающую способность на уровне долей секунд. Первый ее элемент, *tv_sec,* — это число секунд, второй элемент, *tv_nsec,* — число наносекунд в текущей секунде. (Это означает, что никогда не следует устанавливать параметр *tv_nsec* в значение, превышающее 1 миллиард — это будет подразумевать смещение на более чем 1 секунду).

Пример 2. Задание относительного однократного таймера:

```
t_value.tv_sec = 5;
it_value.tv_nsec = 500000000;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 0;
```

Это сформирует однократный таймер, который сработает через 5,5 секунды. (5,5 секунд складывается из 5 секунд и 500,000,000 наносекунд.)

Мы предполагаем здесь, что этот таймер используется как относительный, потому что если бы это было не так, то его время срабатывания уже давно бы его истекло (5.5 секунд с момента 00:00 по Гринвичу, 1 января 1970).

Пример 3. Задание абсолютного однократного таймера:

```
it_value.tv sec = 987654321;
it_value.tv_nsec = 0;
it_interval.tv_sec = 0;
it_iriterval. tv_nsec = 0;
```

Данная комбинация параметров сформирует однократный таймер, который сработает в четверг, 19 апреля 2001 года

В программе могут быть использованы следующие структуры и функции:

TimerCreate(), TimerCreate_r() — Функция создание таймера

```
#include <sys/neutrino.h>
int TimerCreate( clockid_t id,
                 const  struct  sigevent  *event
           );
int TimerCreate_r( clockid_t id,
                 const  struct  sigevent  *event
           );
```

Обе функции идентичны, исключение составляют лишь способы обнаружения ими ошибок:

- TimerCreate() — если происходит ошибка, то возвращается значение -1.
- TimerCreate_r() — при возникновении ошибки ее конкретное значение возвращается из секции Errors.

Struct sigevent — Структура, описывающая событие таймера

```
#include <sys/siginfo.h>
union sigval {
    int         sival_int;
    void       *sival_ptr;
```

```
};
```

Файл <sys/siginfo.h> определяет также некоторые макросы для более облегченной инициализации структуры *sigevent*. Все макросы ссылаются на первый аргумент структуры sigevent и устанавливают подходящее значение sigev_notyfy (уведомление о событии).

SIGEV_INTR — увеличить прерывание. В этой структуре не используются никакие поля. Инициализация макроса: *SIGEV_INTR_INIT(event)*

SIGEV_NONE — Не посылать никаких сообщений. Также используется без полей. Инициализация: *SIGEV_NONE_INIT(event)*

SIGEV_PULSE — посылать периодические сигналы. Имеет следующие поля:

int sigev_coid — ID подключения. По нему происходит связь с каналом, откуда будет получен сигнал;

short sigev_priority — установка приоритета сигналу;

short sigev_code — интерпретация кода в качестве манипулятора сигнала. sigev_code может быть любым 8-битным значением, чего нужно избегать в программе. Значение sigev_code меньше нуля приводит к конфликту в ядре.

Инициализация макроса: *SIGEV_PULSE_INIT(event, coid, priority, code, value)*

SIGEV_SIGNAL — послать сигнал процессу. В качестве поля используется: int sigev_signo — повышение сигнала. Может принимать значение от 1 до -1. Инициализация макроса: *SIGEV_SIGNAL_INIT(event, signal)*

SignalAction(), SignalAction_r() // функция определяет действия для сигналов.

```
#include <sys/neutrino.h>

int SignalAction( pid_t pid,
                  void (*sigstub)(),
                  int signo,
                  const struct sigaction* act,
                  struct sigaction* oact );
```

```
int SignalAction_r( pid_t pid,
                    void* (sigstub)(),
                    int signo,
                    const struct sigaction* act,
                    struct sigaction* oact );
```

Все значения ряда сигналов идут от_SIGMIN (1) до _SIGMAX (64).

Если act не NULL, тогда модифицируется указанный сигнал. Если oact не NULL, то предыдущее действие сохраняется в структуре, на которую он указывает. Использование комбинации act и oact позволяет запрашивать или устанавливать (либо и то и другое) действия сигналу.

Структура *sigaction* содержит следующие параметры:

- *void (*sa_handler)()* — возращает адрес манипулятора сигнала или действия для неполученного сигнала, действие-обработчик.
- *void (*sa_sigaction) (int signo, siginfo_t *info, void *other)* — возвращает адрес манипулятора сигнала или действия для полученного сигнала.
- *sigset_t sa_mask* — дополнительная установка сигналов для изолирования (блокирования) функций, улавливающих сигнал в течение исполнения.
- int sa_flags — специальные флаги, для того, чтобы влиять на действие сигнала. Это два флага: SA_NOCLDSTOP и SA_SIGINFO.
  - SA_NOCLDSTOP используется только когда сигнал является дочерним (SIGCHLD). Система не создает дочерний сигнал внутри родительского, он останавливается через SIGSTOP.
  - SA_SIGINFO сообщает Neutrino поставить в очередь текущий сигнал. Если установлен флаг SA_SIGINFO, сигналы ставятся в очередь и все передаются в порядке очередности.

Добавление сигнала на установку:

```
#include <signal.h>
int sigaddset( sigset_t *set,
               int signo );
```

Функция ***sigaddset()*** — добавляет signo в set по указателю. Присвоение сигнала набору. ***sigaddset()*** возвращает — 0, при удачном исполнении; -1, в случае ошибки;

Функция ***sigemptyset()*** **—** обнуление набора сигналов

```
#include <signal.h>
int sigemptyset( sigset_t *set );
```

Возвращает — 0, при удачном исполнении; -1, в случае ошибки.

6.6. Сетевое взаимодействие

Виртуальные каналы. В системе QNX приложение может взаимодействовать с процессом, выполняющимся на другом компьютере сети, так же как с процессом, выполняющемся на своем компьютере. В самом деле, с точки зрения приложения нет никакой разницы между локальными и удаленными ресурсами.

Такая высокая степень прозрачности обеспечивается благодаря использованию виртуальных каналов, которые являются путями, по которым *Сетевой администратор* передает сообщения и сигналы по всей сети.

Виртуальные каналы (ВК) способствуют эффективному использованию ресурсов во всей сети QNX по нескольким причинам:

– при создании виртуального канала имеется возможность задать работу с сообщениями определенной длины: это означает, что вы можете распределить ресурсы для обработки сообщения. Тем не менее, если потребуется послать сообщение, длина которого превышает максимально заданную, виртуальный канал автоматически изменит установленный максимальный размер буфера в соответствии с длиной передаваемого сообщения;

– если два процесса, находящиеся на разных узлах, взаимодействуют между собой более, чем через один виртуальный канал, виртуальные каналы разделяются во времени, так как между процессами существует только один реальный виртуальный канал. Эта ситуация часто возникает, когда процесс обращается к нескольким файлам удаленной файловой системы;
– если процесс подключается к существующему разделенному виртуальному каналу и запрашивает размер буфера больший, чем тот, который используется в данное время, размер буфера автоматически увеличивается;
– когда процесс завершается, все связанные с ним виртуальные каналы освобождаются.

Виртуальные процессы. Процесс-отправитель отвечает за установку виртуального канала между собой и процессом, с которым устанавливается связь. Для этого процесс-отправитель обычно вызывает функцию `qnx_vc_attach()`. При этом, кроме создания виртуального канала, на каждом конце канала создается виртуальный процесс с идентификатором - VID. Для каждого процесса на обоих концах виртуального канала VID представляет собой идентификатор удаленного процесса, с которым устанавливается связь. Процессы связываются друг с другом посредством VID.

Каждый VID обеспечивает соединение, которое содержит следующую информацию:
– локальный pid;
– удаленный pid;
– удаленный nid (идентификатор узла);
– удаленный vid.

Вряд ли вам придется работать с виртуальным каналом напрямую. Если приложению требуется, например, получить доступ к удаленному ресурсу ввода/вывода, то виртуальный канал создается вызываемой библиотечной функцией `open()`. Приложения непосредственно не участвуют в создании или использовании виртуального канала. Если приложение определяет нахождение обслуживающего его процесса с помощью функции `qnx_name_locate()`, то виртуальный канал создается авто-

матически при вызове функции. Для приложения виртуальный канал просто отождествляется с PID.

Отключение виртуальных каналов. Существует несколько причин, по которым процесс не может осуществлять связь по установленным виртуальным каналам, а именно:

– произошло отключение питания компьютера, на котором выполняется процесс;
– был отсоединен кабель сети от компьютера;
– был завершен удаленный процесс, с которым установлена связь.

Любая из этих причин может препятствовать передаче сообщений по виртуальному каналу. Необходимо фиксировать эти ситуации для выполнения приложением необходимых действий с целью корректного завершения работы, в противном случае, отдельные ресурсы могут оказаться постоянно занятыми.

На каждом узле Администратор процессов проверяет целостность виртуального канала. Это делается следующим образом:

1. Каждый раз при успешной передаче по виртуальному каналу, обновляется временная метка, связанная с данным виртуальным каналом, для фиксации времени последней активности;
2. Через интервалы времени, устанавливаемые при инсталляции, Администратор процессов просматривает каждый виртуальный канал. В том случае, если в виртуальном канале нет активности, Администратор процессов посылает сетевой пакет проверки целостности канала Администратору процессов другого узла;
3. В том случае, если ответ не получен, или зафиксирован сбой, виртуальный канал помечается, как сбойный. Далее предпринимается ряд действий, определенных при инсталляции для восстановления связи;
4. Если попытки восстановления закончились безуспешно, виртуальный канал «отключается». Все процессы, блокированные на данном канале, переходят в состояние ГОТОВ (READY). (Процессы анализируют возвращаемый код сбоя виртуального канала.)

Для управления параметрами, связанными с проверкой целостности виртуального канала, используется утилита `netpoll`.

6.7. Первичная обработка прерываний

Как бы нам этого не хотелось, но компьютер не может иметь бесконечное быстродействие. В системе реального времени крайне важно, использовать все циклы работы центрального процессора. Также важно минимизировать интервал времени между возникновением внешнего события и фактическим началом выполнения программы, реализующей ответную реакцию на это событие. Это время называется задержкой или временем ожидания (latency). В системе QNX можно определить несколько типов задержки.

Задержка прерывания — это интервал времени между приемом аппаратного прерывания и началом выполнения первой команды обработчика данного прерывания. В системе QNX все прерывания открыты все время, поэтому задержка прерывания обычно незначительна. Но некоторые критические программы требуют, чтобы на время их выполнения прерывания были закрыты. Максимальное время закрытия прерывания обычно определяет худший случай задержки прерывания; следует отметить, что в системе QNX это время очень мало.

На рис. 6.6 представлена диаграмма обработки аппаратного прерывания соответствующим обработчиком прерываний. Обработчик прерываний либо просто возвращает управление процессу, либо возвращает управление и вызывает «срабатывание» proxy. Времена обработки для разных процессоров различны.

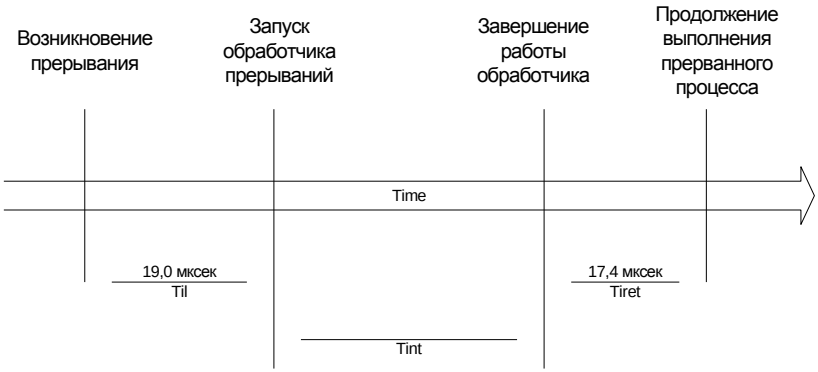

Рис. 6.6. Диаграмма обработки аппаратного прерывания
Til — время задержки прерывания;
Tint — время обработки прерывания;
Tiret — время завершения прерывания.

На диаграмме, приведенной выше, задержка прерывания Til представляет собой минимальную задержку, для случая, когда во время возникновения прерывания все прерывания были открыты. В худшем случае задержка равна этому времени плюс наибольшее время работы процесса QNX, когда прерывания закрыты.

Задержка планирования. В некоторых случаях низкоприоритетный обработчик аппаратных прерываний должен планировать выполнение высокоприоритетных процессов. В этом случае обработчик прерываний возвращает управление и вызывает срабатывание proxy. Это и есть вторая форма задержки — задержка планирования, которую мы рассмотрим ниже.

Задержка планирования — это время между завершением работы обработчика прерываний и началом выполнения первой команды управляющего процесса. Обычно это интервал времени, который требуется для сохранения контекста процесса, выполняющегося в данный момент времени, и восстановления контекста управляющего процесса. Несмотря на то, что это время больше задержки прерывания, оно также остается небольшим в системе QNX.

На рис. 6.7 представлена диаграмма задержки планирования. Обработчик прерываний завершает работу и инициирует

срабатывание proxy. Времена обработки для разных процессоров различны.

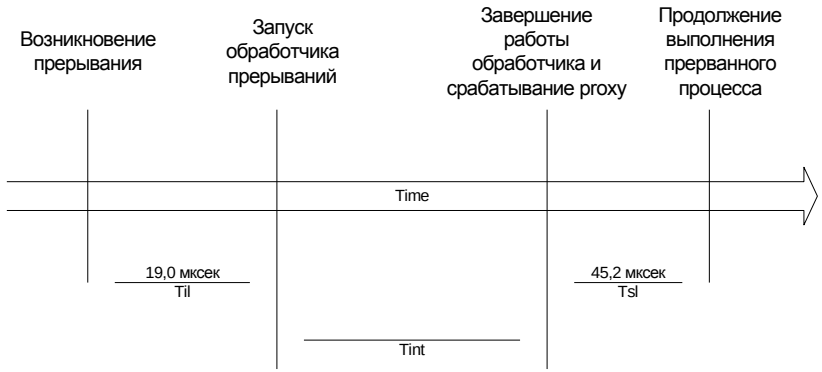

Рис. 6.7. Диаграмма задержки планирования

Til — время задержки прерывания;
Tint — время обработки прерывания;
Tsl — время задержка планирования.

Важно заметить, что большинство обработчиков прерываний завершают работу без инициирования «срабатывания» proxy. В большинстве случаев обработчик прерываний сам справляется со всеми аппаратными событиями. Выдача proxy для подключения управляющего процесса более высокого уровня происходит только при возникновении особых событий. Например, обработчик прерываний драйвера устройства с последовательным интерфейсом, передающий один байт данных аппаратуре, должен на каждое принятое прерывание на передачу запустить высокоуровневый процесс (Dev) только в том случае, если выходной буфер в итоге окажется пустым.

Вложенные прерывания. Поскольку архитектура микрокомпьютера позволяет присваивать аппаратным прерываниям приоритеты, то высокоуровневые прерывания могут вытеснять низкоуровневые. Этот механизм полностью поддерживается в системе QNX.

В предыдущих примерах описаны простейшие и наиболее обычные ситуации, когда поступает только одно прерыва-

ние. Практически такие же временные характеристики имеют прерывания с высшим приоритетом. При расчете наихудших временных показателей для низкоприоритетных прерываний следует учитывать время обработки всех высокоприоритетных прерываний, так как в системе QNX высокоприоритетные прерывания вытесняют низкоприоритетные.

6.8. Диагностическая версия микроядра

Диагностическая версия микроядра ОС QNX Neutrino 6.3. (`procnto-instr`) оснащена сложным механизмом трассировки и протоколирования, который позволяет оценить производительность системы в реальном времени. Модуль `procnto-instr` работает как на однопроцессорных, так и многопроцессорных системах.

Модуль `procnto-instr` вносит весьма небольшие накладные расходы и имеет исключительно высокую производительность, которая, как правило, составляет около 98% производительности обычного ядра с отключенным протоколированием. Можно сказать, что дополнительный объем программного кода диагностического ядра (около 30 Кбайт на архитектуре x86) — сравнительно небольшая цена за увеличение производительности и гибкости благодаря этому весьма полезному инструменту. В зависимости от требований на максимальный объем памяти для итоговой системы, эту специальную версию ядра можно использовать либо как инструмент разработки/создания прототипа, либо как рабочее ядро вашего конечного продукта.

Диагностический модуль не требует изменения исходного кода программы для мониторинга взаимодействия этой программы с ядром. Трассировать можно сколько угодно взаимодействий между ядром и любым активным потоком или процессом, — например, вызовы ядра, изменения состояний и другие системные действия, в том числе даже прерывания, — в этом смысле все эти взаимодействия можно назвать событиями.

Приведем основные задачи, которые выполняет диагностическая версия микроядра:

– Диагностическое ядро (`procnto-instr`) генерирует трассировочные события по различным системным операциям. Эти события автоматически копируются в набор буферов, группированных в циклическом связном списке.
– Как только количество событий в буфере достигает некоторого предела, ядро отправляет извещение соответствующей утилите сбора данных.
– Утилита сбора данных записывает трассировочные события из буфера в устройство вывода (например, последовательный порт, файл данных о событиях и т. д.).
– Затем специальный инструмент выполняет интерпретацию полученных данных и отображает их для пользователя.

В реальной системе производится огромное количество действий, поэтому количество событий, генерируемых ядром, может быть чрезвычайно большим (с точки зрения объема данных и ресурсов для их обработки и хранения). Но все эти данные можно легко контролировать с помощью:
– управления условиями генерации событий;
– применения фильтров ядра для динамического управления генерацией событий;
– реализации собственных обработчиков событий для расширенной фильтрации.

Данные, собранные специальной утилитой ***tracelogger***, могут быть проанализированы либо в реальном времени, либо автономно (по завершении регистрации всех нужных событий). Инструмент системного анализа (System Analysis tool, начиная с версии 6.3, он называется System Profiler), входящий в состав IDE, служит для графического представления этих данных. Это позволяет пользователю «увидеть», что именно происходит в системе.

Кроме применения различных фильтров для отслеживания потока событий, вы также можете задать один из следующих двух режимов, которые ядро использует для генерации событий:
– быстрый (fast) — генерируется только самая важная информация о событии (например, только два аргумента вызова ядра);
– расширенный (wide) — генерируется более подробная информация о событии (например, все аргументы вызова ядра).

Выбор между этими двумя режимами связан с предпочтением скорости или детальности — быстрый режим генерирует меньше данных, а расширенный режим генерирует значительно больше данных по каждому событию. В любом случае генерацию событий в системе можно легко изменять, так как оба режима допускают настройку по каждому отдельному событию.

Вместо того чтобы всегда передавать трассировочные события на внешнее устройство, ядро может хранить все эти события во ***внутреннем циклическом буфере.***

При необходимости этот буфер может выводить свое содержимое на внешнее устройство, когда возникают заданные условия, что делает его мощным инструментом для выявления скрытых ошибок в процессе исполнения.

Данные, характеризующие событие, включают в себя высокоточную временную метку, а также идентификационный номер процессора, который сгенерировал событие. По этой информации можно легко диагностировать сложные проблемы согласования по времени, которые чаще возникают в многопроцессорных системах.

Формат описания события также включает в себя информацию о типе процессорной платформы (например, x86, PowerPC и т. д.) И порядке следования байтов, что упрощает выполнение дистанционного анализа (как в реальном времени, так и автономно). С помощью интерпретатора данных выводимую информацию можно отображать в разных представлениях, например:

– линейное представление всей системы с указанием временных меток;
– текущее представление только активных потоков/процессов;
– представление событий по каждому процессу /потоку с указанием текущих состояний.

Системный анализ с помощью модуля IDE. IDE-модуль, входящий в состав комплекта инструментов системного анализа (Systet Analysis Toolkit, SAT), может служить в качестве полнофункционального инструмента для измерительного контроля и визуализации результатов обработки.

С помощью IDE-модуля SAT разработчики могут настраивать все события и режимы трассировки, а также включать автоматическую передачу файлов протоколирования на удаленную систему для последующего анализа. В качестве инструмента визуализации, IDE-модуль имеет большой набор фильтров событий и процессов, которые предназначены помочь разработчикам быстро анализировать большие массивы событий и делать из них нужную выборку.

Дополнительные средства трассировки. Диагностическая версия микроядра предоставляет превосходные средства для внешнего контроля и мониторинга процессов, потоков и состояний в системе. Однако дополнительная возможность трассировки состоит в том, что сами приложения можно настроить таким образом, чтобы они влияли на процесс отслеживания событий.

С помощью библиотечной функции TraceEvent() приложения можно заставить вставлять заданные события в общий поток событий. Этот механизм особенно полезен при разработке больших, комплексных, многокомпонентных систем.

Вопросы для самопроверки

1. Для управлоения какими объектами существуют системные вызовы в микроядре ОС QNX?
2. Какие вызовы для управления потокамисуществуют в микроядре ОС QNX?
3. В каких состояниях могут находиться потоки в ОС QNX?
4. Каие алгоритмы планирования реализованы в ОС QNX Neutrino 6.3? Опишите их.
5. Опишите механизм работы с потоками.
6. Какие механизмы синхронизации потоков реализованы в ОС QNX?
7. Какие формы межзадачного взаимодействия реализованы в ОС QNX?
8. Опишите связь между процессами посредствам передачи сообщений.
9. Опишите связь между процессами посредствам передачи сигналов.

10. Как строится управление таймером в ОС QNX?
11. Опишите механизм сетевого взаимодействия в ядре ОС QNX.
12. Опишите механизм первичной обработки прерываний в ядре ОС QNX.
13. Что собой представляет диагностическая версия микроядра ОС QNX?

7. Администратор процессов и управление ресурсами в ОС QNX

7.1. Управление процессами

Администратор процессов тесно связан с ядром операционной системы. Однако, несмотря на то, что он разделяет с ядром одно и то же адресное пространство (единственный из всех системных процессов), он выполняется как обычный процесс. Это означает, что Администратор процессов планируется к выполнению ядром и использует те же примитивы передачи сообщений для взаимодействия с другими процессами.

Администратор процессов отвечает за создание в системе новых процессов и за управление ресурсами, связанными с процессом. Все эти функции реализуются посредством передачи сообщений. Например, создание выполняющимся процессом нового процесса осуществляется посредством посылки сообщения, содержащего подробную информацию о вновь создаваемом процессе. Поскольку передача сообщений распространяется на всю сеть, то можно легко создать новый процесс на другом узле, послав соответствующее сообщение Администратору процессов удаленного узла.

Рассмотрим некоторые функции, которые ОС QNX использует для запуска других процессов [10]:

- `system();`
- `fork();`
- `vfork();`
- `exec();`
- `spawn().`

Какую из этих функций применять, зависит от двух требований: переносимости и функциональности.

`system()` — самая простая функция; она получает на вход одну командную строку, такую же, которую вы набрали бы

в ответ на подсказку командного интерпретатора, и выполняет ее.

Фактически, для обработки команды функция system() запускает копию командного интерпретатора.

`fork()` — порождает процесс, являющийся его точной копией. Новый процесс выполняется в том же адресном пространстве и наследует почти все данные порождающего процесса.

Между тем, родительский и дочерний процесс имеют различные идентификаторы процессов, так как в системе не может быть двух процессов с одинаковыми идентификаторами, а также файловые блокировки, ожидающие сигналы, события тревоги и таймеры.

В дочернем процессе функция `fork()` возвращает ноль, а в родительском процессе идентификатор дочернего процесса.

Пример, использования функции fork():

```
printf("PID родителя равен %d\n", getpid());
if (child_pid = fork()) {
printf("Это родитель, PID сына %d\n",
child_pid);
} else {
printf("Это сын, PID %d\n", getpid());
}
```

`vfork()` — так же порождает процесс. В отличие от функции `fork()` она позволяет существенно сэкономить на ресурсах, поскольку она делает разделяемое адресное пространство родителя. Функция `vfork()` создает дочерний процесс, а затем приостанавливает родительский до тех пор, пока дочерний процесс не вызовет функцию `exec()` или не завершится.

`exec()` — заменяет образ порождающего процесса образом нового процесса. Возврата управления из нормально отработавшего `exec()` не существует, т.к. образ нового процесса накладывается на образ порождающего процесса. В системах стандарта POSIX новые процессы обычно создаются без возвра-

та управления порождающему процессу — сначала вызывается `fork()`, а затем из порожденного процесса — `exec()`.

`spawn()` — создает новый процесс по принципу «отец»-«сын». Это позволяет избежать использования примитивов fork() и exec(), что ускоряет обработку и является более эффективным средством создания новых процессов. В отличие от fork() и exec(), которые по определению создают процесс на том же узле, что и порождающий процесс, примитив spawn() может создавать процессы на любом узле сети.

Примитивы `fork()` и `exec()` изначально определены стандартом POSIX, а примитив `spawn()` первоначально реализован в QNX, а в последующем также введен в POSIX.

При создании процесса с помощью одного из трех описанных выше примитивов, он наследует многое от той программной среды, в которой выполнялся его «родитель». Конкретная информация представлена в таблице 7.1.

Таблица 7.1. Возможности наследования при создании процесса

Что наследуется?	**`fork()`**	**`exec()`**	**`spawn()`**
Идентификатор процесса	Нет	Да	Нет
Открытые файлы	Да	На выбор	На выбор
Блокировка файлов	Нет	Да	Нет
Задержанные сигналы	Нет	Да	Нет
Маска сигнала	Да	На выбор	На выбор
Игнорируемые сигналы	Да	На выбор	На выбор
Обработчик сигналов	Да	Нет	Нет
Переменные среды	Да	На выбор	На выбор
Идентификатор сеанса	Да	Да	На выбор
Группа процесса	Да	Да	На выбор
Реальные идентификаторы	Да	Да	Да

группы и пользователя (UID, GID)			
Эффективные UID, GID	Да	На выбор	На выбор
Текущий рабочий каталог	Да	На выбор	На выбор
Маска создания файлов	Да	Да	Да
Приоритет	Да	На выбор	На выбор
Метод планирования	Да	На выбор	На выбор
Виртуальные каналы	Нет	Нет	Нет
Символические имена	Нет	Нет	Нет
Таймеры реального времени	Нет	Нет	Нет

Каждый процесс проходит следующие четыре фазы:

1. ***Создание.*** Создание процесса заключается в присвоении идентификатора процесса (ID) новому процессу и задании информации, определяющей программную среду нового процесса. Большая часть этой информации наследуется от «родителя» нового процесса.
2. ***Загрузку.*** Загрузка образов процессов выполняется загрузчиком «по цепочке». Загрузчик входит в состав Администратора процессов и выполняется под идентификатором нового процесса. Это позволяет Администратору процессов выполнять другие запросы при загрузке программ.
3. ***Выполнение.*** Как только программный код загружен, процесс готов к выполнению; он начинает конкурировать с другими процессами, стремясь получить ресурсы центрального процессора. Обратите внимание на то, что процессы выполняются конкурентно вместе со своими «родителями». Кроме того, гибель «родителя» не вызывает автоматически гибель порожденных им процессов.
4. ***Завершение.*** Процесс завершается одним из двух способов:

– по сигналу, определяющему процесс завершения;
– при возврате управления по функции `exit()` — явно, либо по функции main() — по умолчанию.

Завершение включает в себя две стадии:

1. Администратор процессов инициирует выполнение завершения «по цепочке». Программа завершения входит в состав Администратора процессов и выполняется под идентификатором завершающегося процесса. При этом выполняется закрытие всех описателей открытых файлов, и освобождаются следующие ресурсы:

- все виртуальные каналы, которые имел процесс;
- вся память, выделенная процессу;
- все символические имена;
- все номера основных устройств (только для администраторов ввода/вывода);
- все обработчики прерываний;
- все proxy;
- все таймеры.

2. После запуска программы завершения к породившему его процессу посылается уведомление о завершении процесса (эта фаза выполняется внутри Администратора процессов).

Если породивший процесс не выдал `wait()` или `waitpid()`, то порожденный процесс становится так называемым «зомби-процессом» и не завершается до тех пор, пока породивший процесс не выдаст `wait()` или не завершится сам.

Для того, чтобы не ждать завершения порожденного процесса, следует либо установить признак `_SPAWN_NOZOMBIE` в функциях `qnx_spawn()` или `qnx_spawn_option()`, либо в функции `signal()` задать для `SIGCHLD` признак `SIG_IGN`. В этом случае порожденные процессы при завершении не становятся «зомби-процессами».

Породивший процесс может ожидать завершения порожденного процесса на удаленном узле. Если процесс, породивший «зомби-процесс» завершается, то освобождаются все ресурсы, связанные с «зомби».

Если процесс завершается по сигналу завершения и при этом выполняется утилита dumper, то формируется дамп образа памяти. Этот дамп можно просмотреть с помощью символьного отладчика.

Процесс всегда находится в одном из следующих состояний:

READY (готов) — процесс может использовать центральный процессор (т.е. он не ждет наступления никакого события);

BLOCKED (блокирован) — процесс находится в одном из следующих состояний блокировки:

SEND-блокирован;

RECEIVE-блокирован;

REPLY-блокирован;

SIGNAL-блокирован;

HELD (задержан) — процесс получил сигнал SIGSTOP. До тех пор, пока он не выйдет из состояния HELD, ему не разрешается использовать центральный процессор. Вывести из состояния HELD можно либо выдачей сигнала SIGCONT, либо завершить процесс по сигналу;

WAIT (ожидает) — процесс выдал `wait()` или `waitpid()` и ожидает информацию о состоянии порожденных им процессов;

DEAD (мертв) — процесс завершен, но не может передать информацию о своем состоянии породившему его процессу, поскольку тот не выдал функцию `wait()` или `waitpid()`. За завершенным процессом сохраняется состояние, но занимаемая память освобождается. Процесс в состоянии DEAD также называют «зомби-процессом».

Определены следующие переходы из одного состояния в другое:

– Процесс посылает сообщение.
– Процесс-получатель принимает сообщение.
– Процесс-получатель отвечает на сообщение.
– Процесс ожидает сообщения.
– Процесс принимает сообщение.
– Сигнал разблокирует процесс.
– Сигнал пытается разблокировать процесс; получатель запрашивает сообщение о захвате сигнала.
– Процесс-получатель принимает сигнал.
– Процесс ожидает завершения порожденного процесса.

– Порожденный процесс завершается, либо сигнал разблокирует процесс.
– Процессу выдан SIGSTOP.
– Процессу выдан SIGCONT.
– Процесс завершается.
– Порождающий процесс ожидает завершения, завершается сам или уже завершен.

Определить состояние конкретного процесса возможно:
– из интерпретатора (Shell) — с помощью утилит ps или sin;
– из программы — с помощью функции qnx_psinfo().

Определить состояние операционной системы в целом возможно:
– из интерпретатора — с помощью утилиты sin;
– из программы — с помощью функции qnx_osinfo().

Утилита ps определена стандартом POSIX, командные файлы с ее использованием являются мобильными. Утилита sin уникальна в QNX, она предоставляет полезную информацию о системе QNX, которую нельзя получить с помощью утилиты ps.

В QNX обеспечивается возможность разработки приложений, представляющих собой набор взаимодействующих процессов. Такие приложения отличаются более высокой степенью параллельной обработки данных, и кроме того, могут распределяться в сети, повышая этим производительность системы.

Однако разбиение приложения на взаимодействующие процессы требует специальных соглашений. Для того чтобы взаимодействующие процессы могли надежно связываться друг с другом, они должны иметь возможность определять идентификаторы (ID) друг друга. Рассмотрим, например, сервер базы данных, который работает с произвольным количеством обслуживаемых процессов (клиентов). Клиенты могут обращаться к серверу в любое время, а сервер всегда должен быть доступен. Каким образом клиенты определяют идентификатор сервера базы данных для того, чтобы послать ему сообщение?

В QNX эта проблема решается путем предоставления возможности присваивать процессам символические имена. В случае одного узла процессы могут зарегистрировать это имя с по-

мощью *Администратора процессов* на том узле, где они выполняются. Другие процессы могут затем получить у Администратора процессов идентификатор процесса, соответствующий этому имени.

В случае работы в сети проблема усложняется, т.к. сервер должен обслуживать клиентов, которые находятся на разных узлах сети. В QNX имеется возможность поддерживать работу, как с глобальными, так и с локальными именами. Глобальные имена доступны во всей сети, а локальные — только на том узле, где они зарегистрированы. Глобальные имена начинаются со знака слэш (/). Например:

qnx локальное имя;
company/xyz локальное имя;
/company/xyz глобальное имя.

Для того чтобы использовать глобальные имена хотя бы на одном из узлов сети, необходимо запустить «определитель имен процессов» (утилита nameloc). Этот процесс содержит записи всех зарегистрированных глобальных имен.

В одно и то же время в сети могут работать до десяти определителей имен процессов. Каждый имеет идентичную копию всех активных глобальных имен. Эта избыточность обеспечивает надежность работы сети, гарантируя работоспособность при одновременном аварийном завершении нескольких определителей имен процессов.

Для регистрации имени процесс-сервер использует функцию Си `qnx_name_attach()`. Для определения имени процесса процесс-клиент использует функцию Си `qnx_name_locate()`.

7.2. Обработчики прерываний

Обработчики прерываний обслуживают прерывания аппаратной части компьютерной системы; реагируют на аппаратные прерывания и управляют на нижнем уровне передачей данных между компьютером и внешними устройствами.

Физически обработчики прерываний формируются как часть стандартного процесса QNX (например, драйвера), но они всегда выполняются асинхронно с процессом, в котором содержатся.

Обработчик прерываний обладает следующими свойствами:

– запускается удаленным вызовом, а не прямо прерыванием (лучше писать его на языке Си, а не на ассемблере);
– выполняется внутри процесса, в который встроен, поэтому имеет доступ ко всем глобальным переменным процесса;
– выполняется только для разрешенных прерываний и приоритетно обслуживает прерывания более высокого уровня;
– не взаимодействует непосредственно с контроллером прерываний (микросхемой 8259). Это делает операционная система.

По одному прерыванию (если это поддерживается аппаратно) могут запускаться несколько процессов. При возникновении физического прерывания каждому обработчику прерываний передается управление. В каком порядке обработчики прерываний разделяют обработку этого прерывания — не определено.

Если вы хотите установить аппаратное прерывание, используйте функцию `qnx_hint_attach()`.

Если вы хотите удалить аппаратное прерывание, используйте функцию `qnx_hint_detach()`.

Можно подключить обработчик прерываний напрямую к системному таймеру таким образом, чтобы обработчик запускался по каждому прерыванию от таймера. Для установки периода используйте утилиту ticksize.

Можно также подключиться к масштабируемому прерыванию от таймера, которое выдается каждые 100 миллисекунд в зависимости от значения ticksize. Эти таймеры являются альтернативой таймерам стандарта POSIX 1003.4 при обработке прерываний нижнего уровня.

7.3. Администраторы ресурсов

Управление ресурсами ЭВМ — одна из главных функций любой ОС. К основным ресурсам, которыми управляет QNX, относятся файловые системы, символьные устройства ввода/вывода (последовательные и параллельные порты, сетевые карты и т. д.) и виртуальные устройства («нуль»-устройство, псевдотерминалы, генератор случайных чисел и т. п.). В QNX поддержка ресурсов не встроена в микроядро и организована с помощью специальных программ и динамически присоединяемых библиотек, получивших название «администраторы ресурсов». Взаимодействие между администраторами ресурсов и другими программами реализовано через четко определенный, хорошо документированный интерфейс файлового ввода/вывода. Хотя, конечно, на самом деле весь обмен построен на базовом механизме QNX-сообщений микроядра Neutrino.

Администратор ресурсов — это прикладная серверная программа, принимающая QNX-сообщения от других программ и, при необходимости, взаимодействующая с аппаратурой [10]. Для связи между программой-клиентом и администратором ресурсов используется механизм пространства имен.

Работу администратора ресурсов легко представить в виде схемы:

1. Выполняется инициализация интерфейса сообщений, при этом создается канал, по которому клиенты могут посылать свои сообщения администратору ресурсов.
2. Регистрируется путевое имя (т. е. зона ответственности) в пространстве имен администратора процессов.
3. Запускается бесконечный цикл по приему сообщений от клиентов.
4. С помощью операторов switch/case выполняется переключение на нужный обработчик для каждого типа сообщения.

7.4. Файловые системы в QNX

Классификация файловых систем в QNX. ОС QNX обеспечивает поддержку различных файловых систем с помо-

щью соответствующих администраторов ресурсов, каждый из которых при запуске регистрирует у администратора процессов зону ответственности в виде путевого имени. Такая реализация позволяет запускать и останавливать любую комбинацию файловых систем динамически.

Файловые системы, поддерживаемые в QNX, можно классифицировать следующим образом [10]:

Образная файловая система (image filesystem) — простая файловая система «только для чтения», состоящая из модуля procnto и других файлов, включенных в загрузочный образ QNX. Этот тип файловой системы поддерживается непосредственно администратором процессов и достаточен для многих встроенных систем. Если же требуется обеспечить поддержку других файловых систем, то модули их поддержки добавляются в образ и могут запускаться по мере необходимости.

RAM — плоская «файловая система», которую автоматически поддерживает администратор процессов. Файловая система RAM основана на использовании ОЗУ и позволяет выполнять операции чтения/записи из каталога /dev/shmem. Этот тип файловой системы нашел применение в очень маленьких встроенных системах, в которых не требуется хранение данных на энергонезависимом носителе и для которых достаточно ограниченных функциональных возможностей (нет поддержки каталогов, жестких и мягких ссылок).

Блочные файловые системы — традиционные файловые системы, обеспечивающие поддержку блок-ориентированных устройств типа жестких дисков и дисководов CD-ROM. К ним относятся файловые системы QNX4, DOS, ext2 и CD-ROM:

– файловая система QNX4 (fs-gnx4.so) — высокопроизводительная файловая система, сохранившая формат и структуру дисков ОС QNX4, однако усовершенствованная для повышения надежности, производительности и совместимости со стандартом POSIX;

– файловая система DOS **(fs-dos.so)** обеспечивает прозрачный доступ к локальным разделам FAT (12, 16, 32), при этом файловая система конвертирует POSIX-примитивы работы с диском в соответствующие DOS-команды. Если эквивалентную операцию

выполнить нельзя (например, создать символьную ссылку), то возвращается ошибка;

– файловая система CD-ROM (fs-cd.so) обеспечивает прозрачный доступ к файлам на компакт-дисках формата ISO 9660 и его расширений (Rock Ridge, Joliet, Kodak Photo CD и Audio);

– файловая система Ext2 **(fs-ext2.so)** обеспечивает прозрачный доступ из среды QNX к Linux-разделам жесткого диска как 0, так и 1 версии.

Flash — не блок-ориентированные файловые системы, разрабатываемые специально для устройств флэш-памяти. Мы обсудим этот тип файловой системы в разделе, посвященном встраиванию QNX.

Network — файловые системы, обеспечивающие доступ к файловым системам на других ЭВМ. К ним относятся файловые системы NFS и CIFS (SMB):

– файловая система NFS (Network File System) обеспечивает клиентской рабочей станции прозрачный доступ через сеть к файлам независимо от операционных систем, используемых файл-серверами. NFS использует механизм удаленного вызова процедур (RPC) и работает поверх TCP/IP;

– файловая система CIFS (Common Internet File System) обеспечивает клиентским станциям прозрачный доступ к сетям Windows, а также к UNIX-системам с запущенным сервером SMB. Работает поверх TCP/IP.

Virtual — особые файловые системы, обеспечивающие специфические функциональные возможности при работе с другими файловыми системами:

– пакетная файловая система, обеспечивающая привычное представление выделенных файлов и каталогов для клиента. Эта файловая система будет подробно рассмотрена ниже;

– Inflator, администратор ресурсов, зона ответственности которого устанавливается ближе к корню файловой системы, предназначенный для динамического разжимания файлов, сжатых утилитой deflate.

Реализация поддержки файловых систем.

Поскольку у некоторых файловых систем, работающих в ОС QNX, много общих черт, то для максимизации повторного использования программного кода файловые системы проекти-

руют как комбинацию драйверов и разделяемых библиотек. Такое решение позволяет существенно сократить количество дополнительной памяти, требуемой при добавлении файловой системы в QNX, поскольку добавляется только код, непосредственно реализующий протокол обмена с данной файловой системой.

Например, если администратор конфигурирования аппаратуры enum-devices обнаружил интерфейс EIDE, то запускается драйвер devb-eide. Этому драйверу для работы необходимо загрузить модуль поддержки блок-ориентированного ввода-вывода io-blk.so, который создает в каталоге /dev несколько блок-ориентированных файлов устройств. По умолчанию они обозначаются hdn (для жесткого диска) и cdn (для CD-ROM), n соответствует физическому номеру устройства. Кроме того, для каждого раздела жестких дисков создается свой блок-ориентированный специальный файл с именем hd*n*t*m*, где *n* — номер устройства, а *m* — тип раздела. Например, для раздела FAT32 на первом жестком диске будет создан файл /dev/hd0t11. Если разделов одного типа несколько, то они нумеруются дополнительно с разделением номеров точкой, например, /dev/hd0t11.l.

Модуль io-bik.so обеспечивает для всех блочных файловых систем буферный кэш, в который помещаются данные при выполнении записи на диск. Это позволяет значительно сократить число операций чтения/записи с физическим диском, т. е. повышает производительность работы файловых систем. Однако критичные с точки зрения надежности функционирования блоки файловой системы (например, информация о структуре диска) записываются на диск немедленно и синхронно, минуя обычный механизм записи.

Для доступа к жестким дискам, компакт-дискам и оптическим дискам драйверу необходимо подгрузить соответствующие модули поддержки общих методов доступа — соответственно cam-disk.so, cam-cdrom. so и/или cam-optical.so.

Для поддержки собственно блочных файловых систем модуль io-blk.so загружает необходимые администраторы файловых систем, так же реализованные в виде динамически присо-

единяемых библиотек. Поддержка блочных файловых систем реализована в модулях администраторов ресурсов:

fs-qnx4.so — файловая система QNX4;

fs-ext2.so — файловая система Ext2;

fs-dos.so — файловая система FAT32;

fs-cd.so — файловая система ISO9660.

Администраторы файловых систем монтируют свои разделы в определенные точки файловой системы. Раздел QNX4, выбранный в процессе загрузки как первичный, монтируется в корень файловой системы — /. Остальные разделы по умолчанию монтируются в каталог /fs. Например, компакт-диск монтируется в точку /fs/cd, а файловая система FAT32 монтируется в точку /fs/hd0-dos.

Для того чтобы программа diskboot могла использовать раздел для поиска базового образа файловой системы QNX, необходимо наличие файла /.diskroot.

Операция по изменению информации на диске выполняется в форме транзакции, поэтому даже при катастрофических сбоях (например, при отключении питания) файловая система QNX4 остается цельной. В худшем случае некоторые блоки могут быть выделены, но не использованы, вернуть эти блоки можно, запустив утилиту chkfsys.

Дерево каталогов, которое пользователь привык видеть на экране, — это виртуальная файловая система, которая была создана и управляется администратором пакетной файловой системы fs-pkg.

7.5. Инсталляционные пакеты и их репозитарии

Создавая ОС QNX Neutrino, разработчики думали и о том, каким образом обеспечить простой и эффективный механизм поставки и инсталляции программного обеспечения для QNX. В настоящее время очень распространенным средством доставки ПО от разработчика к пользователю стал Интернет. Поэтому было принято решение, что файлы, входящие в состав программного продукта для QNX, следует помещать в архив формата TGZ, т. е. все файлы компонуются в один утилитой tar, и этот

файл сжимается утилитой gzip. Такой архив и есть QNX-пакет — файл, имеющий расширение qpk.

Чтобы избавить пользователя от необходимости скачивать весь пакет для того, чтобы узнать информацию о производителе, версии, совместимости с другим программным обеспечением и тому подобного, такого рода информация помещается в отдельный от пакета файл манифеста, имеющий расширение qpm. Одну или несколько пар файлов qpk и qpm можно с помощью тех же tar и gzip объединить в репозитарии qpr. В этом случае информация о содержимом репозитария помещается в отдельном файле-манифесте qrm. Репозитарии с его манифестом можно помещать на любом носителе — CD-ROM, веб-сайте и т. п. Для установки пакетов в операционную систему QNX используется уже знакомый вам инсталлятор QNX Software Installer — программа gnxinstall.

Рассмотренный нами тип репозитария — это репозитарий для хранения готовых к инсталляции пакетов программного обеспечения. В составе ОС QNX есть необходимые инструменты для построения собственных инсталляционных пакетов.

В самой ОС QNX есть два типа инсталляционных пакетов:
/pkgs/base — для базовой системы QNX;
/pkgs/repository — для дополнительного программного обеспечения.

Эти каталоги и то, что в них хранится — реальные объекты файловой системы на диске.

Содержимое репозитариев имеет определенную структуру, позволяющую избежать конфликтов между программным обеспечением разных производителей, версиями, целевыми и хост-платформами. Путевое имя каталога, содержащего тот или иной пакет, имеет стандартизованный формат:
/pkgs/repository/производитель/ продукт/каталог_пакета

Внутри каждого пакета обязательно есть XML-файл MANIFEST, который полностью описывает пакет. Остальные файлы и каталоги организованы таким образом, как они должны располагаться в файловой системе, причем за корневой каталог считается каталог пакета. Важная деталь: файлы и каталоги пакета, специфичные для какой-либо аппаратной архитектуры, располагаются в отдельном каталоге (ppcbe, x86, shle и т. д.).

Такой подход, с одной стороны, позволяет экономить дисковое пространство — платформо-независимый код не дублируется. С другой стороны, на одной ЭВМ можно без конфликтов хранить файловые системы для разных аппаратных платформ, что при сетевой прозрачности QNX обеспечивает широкие возможности для машин с ограниченными ресурсами.

Администратор пакетной файловой системы fs-pkg отображает содержимое файловой системы в таком виде, как мы привыкли это видеть, скрывая пакетную структуру данных. Некоторые каталоги существуют реально: /var, /tmp и другие — они создаются при первом старте ОС после инсталляции. Информация о конфигурации пакетов хранится в файле /etc/system/package/packages.

Идеология пакетов предполагает их неизменность, т. е. пакеты должны быть доступны только для чтения. А что же делать, если требуется модифицировать или вообще заменить какой-либо файл? Возможность таких изменений реализована с помощью каталога /var/pkg/spill. Именно туда помещаются измененные и добавленные файлы, а также записывается информация об «удалении».

Деинсталляция и деинициализация пакетов выполняется программой QNX Software Installer.

7.6. Символьные устройства ввода/вывода

Символьными устройствами ввода/вывода называют такие устройства, которые передают или принимают последовательность байт один за другим, в отличие от блок-ориентированных устройств. Имена администраторов символьных устройств ввода/вывода имеют вид **devc-*.** Обычно в системе имеются следующие символьные устройства [10]:

– консольные устройства (или текстовые консоли);
– последовательные порты;
– параллельные порты;
– псевдотериналы (ptys).

Для максимального использования кода управления символьными устройствами используется статическая библиотека io-char. Эта библиотека управляет потоками данных между при-

ложением и драйвером устройства посредством очередей разделяемой памяти. Каждая очередь работает по принципу FIFO.

Режимы ввода при работе устройств:

– поточный или «сырого ввода» (raw) — наиболее производительный режим, однако io-char не выполняет никаких редактирований принимаемых данных;

– редактируемый (edited) — режим, при котором io-char может выполнять операции редактирования над каждым символом строки. После окончания редактирования строки она становится доступной для обработки прикладным процессом (обычно, после ввода символа возврата каретки CR). Такой режим часто именуют каноническим.

Консольные устройства. Системные консоли управляются процессами-драйверами devc-con или devc-tcon. Совокупность клавиатуры и видеокарты с монитором называют физической консолью.

Консольный драйвер devc-con позволяет запускать на физической консоли несколько терминальных сессий посредством виртуальных консолей. При этом devc-con организует несколько очередей ввода/вывода к io-char через несколько символьных устройств с именами /dev/con1, /dev/con2 и т. д. С точки зрения приложения создается эффект наличия нескольких консолей.

Консольный драйвер devc-tcon представляет собой «облегченную» (т. е. поддерживающую только одиночную консоль с «сырым» вводом) версию драйвера devc-con для систем с ограниченным объемом памяти.

Последовательные устройства. Последовательные каналы ввода/вывода управляются семейством процессов-драйверов devc-ser*. Каждый из драйверов может управлять более чем одним физическим устройством и обеспечивать поддержку нескольких символьных устройств.

Параллельные устройства. Параллельные принтерные порты (до четырех) управляются драйвером-процессом devc-par. Этот драйвер поддерживает только вывод, чтение из устройства /dev/par*n* дает результат, аналогичный чтению из /dev/null.

Псевдотерминалы (ptys). Псевдотерминалы управляются драйверным процессом **devc-pty.** С помощью аргумента при запуске драйвера определяется количество псевдотерминалов.

Псевдотерминал состоит из двух частей: основной (master) драйвер и подчиненное (slave) устройство. Подчиненное устройство обеспечивает для прикладных процессов интерфейс, идентичный обычному POSIX-терминалу. Обычный терминал взаимодействует с аппаратным устройством, а подчиненное устройство псевдотерминала вместо этого взаимодействует с основным драйвером псевдотерминала. Основной драйвер может взаимодействовать с другим процессом. Таким образом, псевдотерминал может использоваться для того, чтобы процесс мог взаимодействовать с другим процессом как с символьным устройством.

Псевдотерминалы как правило используются для создания интерфейса для таких программ, как эмуляторы терминала pterm или telnet.

7.7. Сетевая подсистема QNX

ОС QNX — система изначально сетевая, однако сетевые механизмы, как и все остальное, реализованы в виде дополнительных администраторов ресурсов. Хотя некоторая поддержка сети есть в микроядре — способ адресации QNX-сообщений обеспечивает возможность передачи их по сети. Грубо говоря, микроядру Neutrino все равно, является сообщение сетевым или локальным, что дает QNX мощный механизм поддержки сетевых кластеров.

В QNX реализованы средства поддержки двух протоколов — TCP/IP**,** являющегося промышленным стандартом, и Qnet, «родного» протокола QNX, реализующего концепцию «прозрачной сети». Компьютеры, объединенные в сеть Qnet, фактически представляют собой виртуальную многопроцессорную суперЭВМ.

Структура сетевой подсистемы QNX. QNX является сетевой операционной системой и позволяет организовать эффективные распределенные вычисления. В центре реализации сете-

вой подсистемы QNX Neutrino лежит администратор сетевого ввода/вывода io-net (менеджер Net). Процесс io-net при старте регистрирует префикс-каталог /dev/io-net и загружает необходимые администраторы сетевых протоколов и аппаратные драйверы. Протоколы, драйверы и другие необходимые компоненты (все они реализованы в виде DLL) загружаются либо в соответствии с аргументами командной строки, заданными при запуске **io-net,** либо в любое время командой монтирования mount. Упрощенно структуру сетевой подсистемы можно представить как на рис. 7.1.

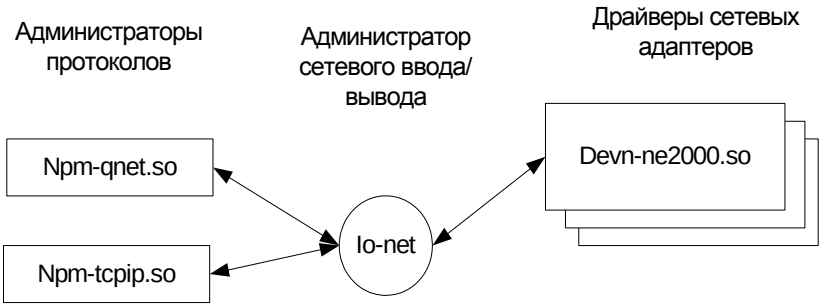

Рис. 7.1. Структура сетевой подсистемы

Следующая команда запускает поддержку сети с драйвером сетевого адаптера NE2000 и с поддержкой протоколов TCP/IP и Qnet: `io-net -dne2000 -ptcpip -pqnet`

Или, например, можно выполнить следующие команды:

1. Запустить администратор сети: `io-net &`
2. Загрузить драйвер Ethernet для адаптера NE2000:

```
mount -T io-net devn-ne2000.so
```

3. Загрузить администратор протокола Qnet:

```
mount -T io-net npm-qnet.so
```

4. Загрузить администратор протокола TCP/IP:

```
mount -T io-net npm-tcpip.so
```

Отключить поддержку, например, Qnet и выгрузить DLL npm-qnet.so можно командой:

```
umount /dev/ io-net /qnet0
```

Для получения диагностической и статистической информации о работе сетевой карты используется утилита nicinfo (Network Interface Card INFOrmation). По умолчанию nicinfo обращается к устройству /dev/io-net/en0. Для адаптеров Ethernet nicinfo выдает наименование модели контроллера и другие характеристики:

– Физический адрес (MAC-адрес) адаптера.
– MAC-адрес, присвоенный адаптеру (имеет смысл для адаптеров с программируемыми MAC-адресами).
– Скорость и способ передачи данных.
– Максимальный размер кадра, байты.
– Логический номер сети (используется, если к ЭВМ подключено несколько сетевых адаптеров).
– Диапазон портов.
– Номер прерывания.
– Признак включения режима приема всех пакетов.
– Признак включения режима групповой передачи.
– Общее число удачно отправленных кадров.
– Общее число кадров отправленных с ошибками.
– Общее число удачно принятых кадров.
– Общее число принятых пакетов, содержащих ошибки.
– Количество отправленных байтов.
– Количество полученных байтов.
– Количество коллизий при передаче.
– Количество коллизий при передаче с отменой передачи пакета.
– Количество потерь при передаче.
– Число попыток.
– Количество пакетов, передача которых была отложена.
– Число поздних коллизий.
– Количество переполнений приемного буфера.
– Число ошибок выравнивания.
– Количество ошибок по несовпадению контрольной суммы.

Для разработки собственных драйверов сетевых карт в состав QNX Momentics входит специальный программный пакет, называемый Network Driver Development Kit (Network DDK), включающий исходные тексты нескольких драйверов и детальную инструкцию по написанию аппаратно-зависимого кода.

QNX-сеть — Qnet. Прозрачность сетевого взаимодействия в сети Qnet основана на способности QNX выполнять передачу сообщений между микроядрами Neutrino через сеть. Каким же образом можно отличить сетевое сообщение от локального и, главное, как идентифицировать узлы сети? Для этой цели служит пространство имен путей администратора процессов. При загрузке и инициализации администратор npm-qnet.so регистрирует символьное устройство /dev/io-net/qnet0 и префикс-каталог /net, в котором помещаются папки с именами узлов сети. То есть, например, если в сети есть узлы с именами alpha и beta, каталог /etc узла beta является каталогом /net/beta/etc узла alpha.

Таким образом, Qnet обеспечивает прозрачный доступ к файлам всех узлов сети с помощью обычных локальных средств — файлового менеджера, команды is и т. п. Кроме того, Qnet позволяет запускать процессы на любом узле сети. Для этого используется утилита on.

Утилита on является своего рода расширением возможностей командного интерпретатора по запуску приложений. Синтаксис утилиты таков:

```
on опции команда
```

При этом команда будет выполнена в соответствии с предписаниями, заданными посредством опций. Укажем основные опции этой утилиты:

-n node — указывает имя узла (node), на котором должна быть запущена команда. При этом в качестве файловойсистемы будет использована файловая система узла, с которого запущен процесс;

-f node — то же, что и предыдущая опция, но в качестве файловой системы будет использована файловая система узла, на котором запущен процесс;

-t tty — указывает, с каким терминалом целевой ЭВМ должен быть ассоциирован запускаемый процесс;

-d — отсоединиться от родительского процесса. Если задана эта опция, то утилита on завершится, а запущенный процесс будет продолжать выполняться (т. е. он будет запущен с флагом SPAWN_NOZOMBIE);

-p приоритет [дисциплина] — можно задать значение приоритета и, если надо, дисциплину диспетчеризации. Например, команда:

```
on -d -n host5 -p 30f -t con2 inetd
```

запустит на консоли /dev/con2 узла с именем hosts программу inetd, находящуюся на нашем узле. Приоритет запущенного процесса будет иметь значение 30 с дисциплиной диспетчеризации FIFO. После запуска программы inetd утилита on сразу завершит свою работу, и интерпретатор выдаст приглашение для ввода очередной команды.

Здесь уместно вспомнить про то, как администратор пакетной файловой системы fs-pkg отображает платформенно-зависимые файлы. Как вы помните, такие файлы отображаются не только в корневой каталог /, но и в платформенно-зависимый каталог, например, /ppcbe. Это позволяет брать на удаленном узле те файлы, которые предназначены для выполнения на аппаратуре конкретного узла сети. Например, мы хотим на нашем бездисковом компьютере на базе процессора PowerPC (Big Endian) запустить утилиту is, расположенную на узле с именем alpha (и с процессором Pentium III). Для этого выполним команду:

```
/net/alpha/ppcbe/usr/bin/ls -I /home
```

Этот механизм дает возможность без конфликтов использовать дисковое пространство всех узлов сети Qnet независимо от аппаратуры, на которой работает QNX Neutrino. В сочетании с возможностями утилиты on администратор получает «виртуальную суперЭВМ». Но не забывайте про «ложку дегтя» — в сети Qnet забудьте о безопасности информации. Информацию о доступных узлах сети Qnet можно посмотреть командой **sin net.** В результате получим список доступных узлов Qnet-сети с информацией о каждом из них:

```
$ sin net
myhost   144M   1   631   Intel   686
F6M8S6
hishost   467M   1   939   Intel   686
F6M8S10
```

Как видно из приведенного фрагмента, выдается имя узла, размер доступной оперативной памяти, количество процессоров, тактовая частота и модель процессора.

Обратите внимание, что некоторые утилиты QNX имеют сетевую опцию, указывающую, к какому узлу сети применить действие утилиты (например, sin или slay).

Протокол транспортного уровня FLEET. Для организации обмена в сети используется надежный и эффективный протокол транспортного уровня FLEET. Каждый из узлов может принадлежать одновременно нескольким QNX-сетям. В том случае, если сетевое взаимодействие может быть реализовано несколькими путями, для передачи выбирается незагруженная и более скоростная сеть.

Сетевое взаимодействие является узким местом в большинстве операционных систем и обычно создает значительные проблемы для систем реального времени. Для того чтобы обойти это препятствие, разработчики QNX создали собственную специальную сетевую технологию FLEET и соответствующий протокол транспортного уровня FTL (FLEET transport layer). Этот протокол не базируется ни на одном из распространенных сетевых протоколов типа IPX или NetBios и обладает рядом качеств, которые делают его уникальным. Основные его качества зашифрованы в аббревиатуре FLEET, которая расшифровывается в таблице 7.2 [8].

Таблица 7.2. Расшифровка аббревиатуры FLEET

Fault-Tolerant Networking	QNX может одновременно использовать несколько физических сетей. При выходе из строя любой из них данные будут автоматически перенаправлены «на лету» через другую сеть.
Load-Balancing on the Fly	При наличии нескольких физических соединений QNX автоматически распараллеливает передачу пакетов по соответ-

Efficient Performance	Специальные драйверы, разрабатываемые фирмой QSSL для широкого спектра оборудования, позволяют с максимальной эффективностью использовать сетевое
Extensable Architecture	Любые новые типы сетей могут быть поддержаны путем добавления соответствующих драйверов.
TransparentDistributed Processing	Благодаря отсутствию разницы между передачей сообщений в пределах одного узла и между узлами нет необходимости вносить какие-либо изменения в приложения для того, чтобы они могли взаимодействовать через сеть.

Благодаря этой технологии сеть компьютеров с QNX фактически можно представлять как один виртуальный суперкомпьютер. Все ресурсы любого из узлов сети автоматически доступны другим, и для этого не нужны специальные «фокусы» с использованием технологии RPC. Это значит, что любая программа может быть запущена на любом узле, при этом ее входные и выходные потоки могут быть направлены на любое устройство на любых других узлах.

Например, утилита `make` в QNX автоматически распараллеливает компиляцию пакетов из нескольких модулей на все доступные узлы сети, а затем собирает исполняемый модуль по мере завершения компиляции на узлах. Специальный драйвер, входящий в комплект поставки, позволяет использовать для сетевого взаимодействия любое устройство, с которым может быть ассоциирован файловый дескриптор, например последовательный порт, что открывает возможности для создания глобальных сетей.

Достигаются все эти удобства за счет того, что поддержка сети частично обеспечивается и микроядром (специальный код в его составе позволяет QNX фактически объединять все микроядра в сети в одно ядро).

Когда ядро получает запрос на передачу данных процессу, находящемуся на удаленном узле, он переадресовывает этот за-

прос менеджеру Net, в подчинении которого находятся драйверы всех сетевых карт. Имея перед собой полную картину состояния всего сетевого оборудования, менеджер Net может отслеживать состояние каждой сети и динамически перераспределять нагрузку между ними. В случае, когда одна из сетей выходит из строя, информационный поток автоматически перенаправляется в другую доступную сеть, что очень важно при построении высоконадежных систем. Кроме поддержки своего собственного протокола, Net обеспечивает передачу пакетов TCP/IP, SMB и многих других, используя то же сетевое оборудование. Производительность QNX в сети приближается к производительности аппаратного обеспечения.

Поддержка TCP/IP в QNX. Поддержка стека протоколов TCP/IP обеспечивается с помощью трех модулей, которые могут загружаться администратором сетевого ввода/вывода io-net:

– /lib/dll/npm-ttcpip.so — облегченный стек TCP/IP для систем с ограниченными ресурсами, реализует часть функциональности полных реализаций TCP/IP стека;

– /lib/dll/npm-tcpip-v4.so — стандартная реализация стека протоколов NetBSD vl.5;

– /lib/dll/npm-tcpip-v6.so — профессиональный стек прото колов TCP/IP, KAME-расширение стека NetBSD vl.5.

Файл npm-tcpip.so — это просто ссылка на один из указанных модулей. Для обеспечения безопасности в модуле профессионального стека TCP/IP используется IPSec, который защищает IP-пакет от несанкционированного изменения посредством присоединения криптографической контрольной суммы, вычисленной односторонней хэш-функцией, или/и шифрует содержимое пакета криптографическим алгоритмом с закрытым ключом.

Для пакетной фильтрации (firewall) и трансляции сетевых адресов (NAT) используется перенесенный в QNX вариант IP Filter 3.4.27. Основу пакета IP Filter составляет модуль ipfilter.so, загружаемый администратором io-net. В состав IP Filter также входит ряд утилит:

ipf — утилита для изменения списка правил фильтрации;

ipfs — сохраняет и восстанавливает информацию для NAT и таблицы состояний;
ipstat — возвращает статистику пакетного фильтра;
ipmon — монитор для сохраненных в журнале пакетов; ipnat — пользовательский интерфейс для NAT.

Конфигурация сети TCP/IP хранится в файле /etc/net.cfg. Для того чтобы изменения, сделанные в файле, вступили в силу, необходимо запустить администратор конфигурирования TCP/IP — утилиту netmanager. Обычно для изменения настроек TCP/IP используют графическую утилиту phlip, которая автоматически запускает netmanager при сохранении изменений.

Для получения информации о TCP/IP в QNX содержится полный набор общепринятых UNIX-утилит.

7.8. Технология Jump Gate

Для обеспечения прозрачности в графической оболочке Photon существует механизм, получивший название Jump Gate Technology.

Технология Jamp Gate основана на использовании серверного процесса **phrelay,** который передает клиентским программам информацию о графическом изображении в Photon. Клиентами **phrelay** могут быть: **phditto, phindows, phinx.** Подключиться к серверу можно либо через последовательный канал, либо через сеть TCP/IP. Для запуска phrelay в TCP/IP обычно используется **inetd. B** стандартном файле /etc/ineta.conf уже есть (в закомментированном виде) нужная запись, поэтому достаточно просто раскомментировать ее:

```
phrelay stream tcp nowait root /usr/bin/phrelay phrelay
```

Проверьте, чтобы файл /etc/services содержал строку:

```
phrelay 4 8 6 8/tcp
```

Программы-клиенты кэшируют получаемую информацию, поэтому им достаточно получать данные только об изменениях «картинки».

При удаленном подключении к Photon microGUI по умолчанию создается дополнительная сессия. Чтобы подключиться к существующей сессии, необходимо указать имя «Named Special

Device»-файла нужной сессии. Например, подключимся к текущей сессии Photon узла host1:

```
phditto -n /dev/photon hostl
```

Для доступа к phrelay из Windows используется клиентская программа Phindows.

Для доступа к phrelay из ЭВМ, использующей графическую среду X Window System независимо от аппаратной платформы и от операционной системы, используется QNX-утилита phinx.

Сервер **phrelay** может подключить клиента как к существующей сессии photon (тогда клиент получит полный доступ к запущенным приложениям), так и к отдельной, специально созданной сессии (тогда пользователи будут работать независимо в разных окружениях). Запретить или разрешить подключение к своей графической среде пользователь может, установив или сбросив флаг в окне утилиты **phrelaycfg**.

Для того что чтобы передавать не все изображение рабочего стола, а **только** окно нужного приложения, можно использовать так называемые сервисы **phrelay.** Сервисы определяются в файле конфигурации `/etc/config/phrelay.имя узла` (если такого файла нет, то используется /etc/config/phrelay). Формат этого файла определен так:

```
service user [-W pwm_options] command
```

При этом:

– service — это произвольное символическое имя сервиса, которое передается клиентской утилите с опцией -s;

– user — определяет, как интерпретировать аргумент, передаваемый клиенту с опцией -u;

– pwm_options — необязательные опции оконного менеджера pwm;

– command — командная строка для запуска приложения Photon
(ради которого, собственно, и затевалось все дело).

Особого внимания стоит, пожалуй, параметр user. Он может принимать такие значения:

– имя_пользователя (имеет смысл, если имя пользователя не задано посредством опции -u). Если для пользователя существует пароль, то будет запрошен ввод пароля;

– имя_пользователя:пароль (имеет смысл, если имя пользователя не задано посредством опции -u). Пароль не запрашивается;
– % — будут запрошены имя и пароль (имеет смысл, если не использована опция -u);
– ? — имя и пароль будут запрашиваться всегда (опция -u игнорируется);
– ! — возвращает ошибку, если не задана опция -u;
– =имя_пользователя — имя пользователя задается жестко (опция -и игнорируется);
– =имя__пользователя:пароль — имя пользователя и пароль за-даются жестко (опция -u игнорируется).

Пример СТРОКИ файла /etc/config/phrelay:

```
mylable ivan pfm
```

Теперь, выполнив команду `phindows -smylable`, мы получим такое окно, запущенное от имени пользователя «ivan». Использование сервисов phrelay снижает сетевой трафик.

7.9. Графический интерфейс пользователя Photon microGUI

По аналогии с операционной системой QNX, графическая среда Photon microGUI представляет собой графическое микроядро с семейством процессов, расширяющих функциональность графического микроядра (или «графического сервера») и взаимодействующих посредством стандартного QNX-механизма передачи сообщений. Само графическое микроядро Photon представляет собой небольшой процесс, реализующий только несколько фундаментальных примитивов, которые используются внешними процессами для выполнения различных задач пользовательского интерфейса.

Сервер Photon не работает с окнами, не взаимодействует с устройствами ввода (мышью, клавиатурой и т. п.). За непосредственную прорисовку изображения отвечает *графический драйвер*. Задача графического драйвера — отображение информации, получаемой от сервера шрифтов и от интерпретато-

ра графического *потока.* За ввод информации от мыши, клавиатуры и других устройств отвечает *драйвер ввода.*

Следует заметить, что Photon наследует сетевую прозрачность ОС QNX — дополнительные графические драйверы могут использоваться для расширения графического пространства Photon за счет физических дисплеев других узлов сети. При этом можно легко обеспечить дублирование изображения.

Photon использует кодировку Unicode, что обеспечивает ввод и вывод текста, написанного на разных языках.

Для того чтобы обеспечить полнофункциональную графическую среду, позволяющую пользователям манипулировать окнами приложений, изменением их размеров, перемещением, сворачиванием и т. п., используется *оконный менеджер.* Он же поддерживает панель задач. Рабочий стол с меню быстрого запуска приложений реализован с помощью процесса *администратора рабочего стола.*

Из сказанного видно, что полная функциональность графического интерфейса пользователя ОС QNX достигается набором процессов и DLL, расширяющих возможности графического сервера Photon. Поэтому легко конфигурировать системы с ограниченными ресурсами — нужно просто выбрать только необходимые компоненты. Кроме того, облегчается перенос драйверов видеокарт и устройств ввода.

Реализация графической среды. В обычной настольной системе Photon может запускаться одним из двух способов.

– Вручную из командной строки в любое время после регистрации в системе.

– Автоматически утилитой tinit, если не существует файла `/etc/config/system/nophoton`.

В любом случае используется командный сценарий /usr/bin/ph, выполняющий запуск всех необходимых компонентов графической среды в зависимости от конфигурации системы. Вот что делает этот сценарий:

1. Запускает утилиту зондирования графического оборудования crttrap; если такое оборудование найдено и опознано, то crttrap запускает программу devgt-iographics для определения доступных графических режимов видеокарты. Программа devgt-

iographics записывает результаты своей работы в файл /etc/system/config/graphics-modes.

2. Запускается графический сервер Photon. Если существует переменная окружения LOGNAME (а это означает, что вы ее либо нарочно инициализировали в командных скриптах, либо вы уже прошли регистрацию с помощью утилиты login), то Photon стартует сразу. В противном случае Photon запустит утилиту phiogin для регистрации пользователя в системе. Можно запретить пользователю выход из Photon в командную строку, присвоив переменной PHEXIT_DISABLE значение 1.

3. Программа crttrap запускает администратор графического вывода io-graphics, используя командную строку из файла /etc/system/config/graphics-modes. Процесс io-graphics загружает интерпретатор графического потока gri-photon.so и сервер шрифтов. По умолчанию io-graphics загружает сервер шрифтов, реализованный в виде разделяемого объекта — pnfont.so. Можно указать администратру графического вы вода запускать сервер шрифтов в виде отдельного процесса phfont, создав переменную окружения PHFONT_USE_EXTERNAL. Отдельный процесс сервера шрифтов может понадобиться для того, чтобы его можно было использовать с других узлов сети. Заметим, что в дистрибутиве QNX есть несколько серверов шрифтов с различными ограничениями функциональности для встраиваемых систем.

4. Запускается процесс inputttrap для зондирования устройств ввода. Он определяет, с какими аргументами необходимо запускать драйвер-администратор графического ввода devi-hirun и запускает его. Результат своей работы inputttrap сохраняет в файле `/etc/config/trap/input.имя_узла`.

5. Запускается утилита fontsieuth, указывающая серверу шрифтов phfont, в каких каталогах находятся шрифты.

6. Затем запускаются процессы bkgdmgr (рисует фон рабочего стола), wmswitch (позволяет переключаться между открытыми окнами приложений, используя комбинацию клавиш «Alt»-«Tab»), saver («хранитель экрана»). Кроме того, сам сервер Photon запускает оконный менеджер pwm и администратор рабочего стола shelf.

Утилиты конфигурирования. Компоненты Photon можно конфигурировать с помощью нескольких утилит. Для того чтобы изменить настройки оконного менеджера используется утилита.

Вкладка Background утилиты pwmopts позволяет настраивать параметры администратора фона bkgdmgr.

Для настройки подсистемы шрифтов предназначена утилита fontadmin.

Эта программа позволяет задавать необходимые псевдонимы для инсталлированных шрифтов. Настройка администратора графического вывода io-grafics выполняется с помощью программы phgrafx.

В окне этой утилиты можно задать, какой видеодрайвер следует использовать, значения разрешения, глубины цветности и частоты обновления экрана.

Настройка мыши может выполняться с помощью утилиты input-cfg.

Разрешить либо запретить удаленное подключение к локальной сессии Photon можно с помощью утилиты phrelaycfg.
Установить необходимый хранитель экрана, время отсутствия сигналов ввода, через которое срабатывает хранитель экрана, пароль хранителя можно с помощью утилиты savecfg.

7.10. Печать в ОС QNX

Традиционная система печати (lpd). Эта система широко известна в UNIX-подобных ОС. И, как следствие, достаточно хорошо документирована. Поэтому мы сделаем лишь краткий ее обзор.

Система печати lpd условно делится на три части:

1. Сервер печати (спулер) lpd.
2. Файл Конфигурации Принтеров /etc/printcap.
3. Клиентские утилиты lpr, lprq, lprrm, lprc.

Самой многофункциональной из всех утилит системы lpd является lprc. Она позволяет выполнять все то, что делают другие утилиты, плюс выполняет активизацию/деактивизацию принтеров, управляет очередью и делает другую полезную работу.

Утилита lpr выполняет постановку задания в очередь для печати. Утилитой lprq можно просматривать очередь заданий. Удалить задание из очереди можно утилитой lprrm.

Данная система достаточно старая и многие поставщики ОС предлагают более удобные решения.

Собственная система печати QNX. Основу системы печати QNX является серверный процесс-администратор spooler. Назначение spooler — это обеспечение бесконфликтного доступа нескольких пользователей к контролируемому им устройству. Этот процесс автоматически запускается администратором нумерации устройств enum-devices. По умолчанию spooler контролирует доступ к параллельному порту /dev/par1. Если к параллельному порту подключить принтер, то spooler автоматически распознает его и путем зондирующих запросов получает параметры принтера.

В каталоге /etc/printers содержится несколько файлов конфигурации spooler для разных типов принтеров. При запуске spooler выполняет несколько операций:

– Определяет тип принтера и выбирает соответствующий этому типу файл конфигурации.
– Сравнивает свойства, указанные в файле конфигурации, с параметрами принтера, полученными при его сканировании.
– Регистрирует в пространстве имен префикс-каталогов /dev/printers/имя_принтера/. В этом каталоге создаются каталог spool/, файлы устройств phs, raw и файл, соответствующий типу принтера.
– Создает каталог /var/spool/printers/имя_принтера.имя_хоста/. В этот каталог отображается содержимое /dev/printers/имя_принтера/spool.
– В каталог spool записывается файл с результирующими настройками принтера.

Когда пользователь выдает задание на печать, соответствующий файл помещается в каталог spool, и spooler вызывает необходимые фильтры для обработки файла. Результатом обработки является файл в формате, понятном принтеру. Конечный файл посылается в устройство печати.

В составе дистрибутива ОС QNX поставляется несколько фильтров для наиболее популярных наборов принтеров. Кроме того, в состав ОС QNX входит Printer DDK (Driver Development Kit), представляющий собой подробно комментированный пример исходного кода фильтра с инструкцией для разработчиков.

Для управления заданиями можно воспользоваться фотоновской утилитой prjobs.

Вопросы для самопроверки

1. Какие механизмы существуют в ОС QNX для создания процессов?
2. Какие фазы проходит каждый процесс?
3. В каких состояниях могут находиться процессы?
4. Какими свойствами обладает обработчик прерываний?
5. Назовите назначение администратора ресурсов в ОС QNX.
6. Приведите классификацию файловых систем ОС QNX.
7. Расскажите о реализации файловых систем в ОС QNX.
8. Для чего используются инсталляционные пакеты и репозитарии в ОС QNX?
9. Какие символьные устройства ввода/вывода существуют в QNX?
10. Приведите структуру сетевой подсистемы ОС QNX.
11. Какие сетевые протоколы поддерживаются ОС QNX?
12. Что представляет собой технология FLEET?
13. Опишите технологию Jump Gate.
14. Как в ОС QNX реализован графический интерфейс пользователя.
15. Опишите процесс печати в ОС QNX.

Глоссарий

ABI — Application Binary Interface;
APEX — Application/Executive;
API — Application Program Interface;
ARINC — Avionics Application Software Standard Interface;
BKL — Big Kernel Lock;
BSP — Board Support Package;
CAMAC — Computer Application for Measurement and Control;
COOL — Chorus Object-Oriented Layer;
DLL — Dynamically Linked Libraries;
EAL — Evaluation Assurance Levels;
GRT — Generic Run-Time;
IPC — Inter-Process Communication;
LAP — Local Access Point;
PLC — Programming Logical Controller;
POSIX — Portable Operating System Interface for Computer Environments;
RPC — Remote Procedure Call;
RTAPI — Real-Time Application Program Interface;
RTCA — Radio Technical Commission for Aeronautics;
RTOS — Real Time Operating System;
RTSS — Real-Time Subsystem;
SCADA — Supervisory Control And Data Acquisition;
SCEPTRE — Standardisation du Cœur des Exécutifs des Produits Temps Réel Européens;
SCPI — Standart Commands for Programmable Instruments;
SCSI — Small Computer System Interface;
SMP — Symmetric Multiprocessing;
TCSEC — Trusted Computer System Evaluation Criteria;
VDX — Vehicle Distributed eXecutive;
VICbus — VME Interconnect bus;
VMEbus — Versa Module Eurocard bus;
VMM — Virtual Machine Manager;
VXIbus — VME eXtention for Instruments bus;

АРМ — Автоматизированное рабочее место;
АСУТП — Автоматизированная система управления технологическим процессом;
ОЗУ — Оперативно запоминающее устройство;
ОС — Операционная система;
ОСРВ — Операционная система реального времени;
ПЗУ — Постоянно запоминающее устройство;
САР — Система автоматического регулирования;
САУ — Система автоматического управления;
СОИ — Система отображения информации;
СРВ — Система реального времени;

Список литературы

1. Д. Бэкон, Т. Харрис. Операционные системы — СПб.: Питер; Киев: Издательская группа BHV, 2004. — 800с.: ил.
2. Верхалст Э. Задача разработки ОСРВ для цифровой обработки сигналов // Мир компьютерной автоматизации. — 1997., №4.
3. Timmerman M., Van Beneden B., Uhres L. RTOS Evaluation Kick Off! // Real-Time Magazine. — 1998., №3, С. 6-10.
4. Жданов А.А. Операционные системы реального времени, ЗАО "РТСофт", Москва, "PCWeek", N 8, 1999
5. Алексеев Д. Практика работы с QNX / Д. Алексеев, Е. Ведревич, А. Волков и др. — М.: Издательский Дом «КомБук», 2004. — 432 с.
6. И.Б. Бурдонов, А.С. Косачев, В.Н. Пономаренко Операционные системы реального времени // Препринт Института системного программирования РАН — Открытый электронный ресурс http://www.citforum.ru/operating_systems/rtos/
7. VSPWorks. The RTOS for DSP and ASIC Cores // Wind River technical brief — Открытый электронный ресурс http://www.eonic.co.kr/data/datasheet/windriver/VSPWorks_Technical_Brief.pdf
8. Гордеев А.В., Молчанов. А.Ю. Системное программное обеспечение — СПб.: Питер, 2002. — 736с.
9. Операционная система реального времени QNX Neutrino 6.3. Системная архитектура: Пер. с англ. — БХВ-Петербург, 2005. — 336 с.: ил.
10. Зыль С.Н. Операционная система реального времени QNX: от теории к практике. — СПб.: БХВ-Петербург, 2004. — 192с.: ил.